JAVA

Iniciación



Luis Minguillón Pascual



Luis Minguillón Pascual

Prefacio

Desde siempre he creído que el conocimiento es un bien público, un faro que ilumina el camino hacia la libertad y el crecimiento personal. Esta convicción ha sido la fuerza motriz que me ha llevado a escribir este libro de programación en Java. Mi intención no es solo compartir las herramientas y técnicas necesarias para dominar este lenguaje, sino también transmitir la idea de que aprender y enseñar son actos que nos liberan y nos conectan como comunidad.

En un mundo donde la información a menudo se vuelve inaccesible o está limitada por barreras económicas o sociales, quiero contribuir con un recurso abierto, claro y accesible para todos aquellos que deseen adentrarse en el apasionante mundo de la programación. Creo firmemente que al compartir el saber, fomentamos la igualdad de oportunidades y ayudamos a construir un futuro más justo.

Este libro no hubiera sido posible sin el apoyo incondicional de mi familia. Quiero expresar mi más profundo agradecimiento a todos ellos, y muy especialmente a mi esposa Marisa, cuyo amor, paciencia y respaldo constante me acompañaron en cada paso del proceso de creación de este proyecto. Su comprensión y ánimo fueron esenciales para convertir esta idea en realidad.

Espero que este libro sea para ti una guía útil y una fuente de inspiración en tu aprendizaje. Que te ayude no solo a dominar Java, sino también a valorar el poder transformador del conocimiento compartido.

Con gratitud,
Luis Minguillón Pascual

Indice

Prefacio

- **Capítulo 1: Introducción al lenguaje Java**
    - 1.1 Historia y evolución de Java
    - 1.2 Características principales del lenguaje
    - 1.3 Tipos de aplicaciones Java
    - 1.4 Instalación del JDK y configuración del entorno
    - 1.5 Primer programa en Java
    - 1.6 Uso de la línea de comandos y compilación
- **Capítulo 2: Fundamentos del lenguaje Java**
    - 2.1 Tipos de datos primitivos
    - 2.2 Variables y constantes
    - 2.3 Operadores aritméticos, lógicos y relacionales
    - 2.4 Conversión de tipos
    - 2.5 Estructuras de control (if, switch, while, for, do-while)
    - 2.6 Expresiones booleanas y operadores ternarios
    - 2.7 Sentencias break, continue y return
- **Capítulo 3: Programación Orientada a Objetos en Java**
    - 3.1 Clases y objetos
    - 3.2 Constructores
    - 3.3 Atributos y métodos
    - 3.4 Encapsulamiento
    - 3.5 Modificadores de acceso
    - 3.6 Herencia
    - 3.7 Polimorfismo
    - 3.8 Sobrecarga de métodos y constructores
    - 3.9 Clases abstractas
    - 3.10 Interfaces
    - 3.11 Uso de This y Super
    - 3.12 Enumeraciones
- **Capítulo 4: Manejo de Cadenas de Texto en Java**
    - 4.1 La Clase String
    - 4.2 Métodos útiles de String
    - 4.3 StringBuilder y StringBuffer
    - 4.4 Comparación y manipulación de cadenas
- **Capítulo 5: Arreglos y Estructuras de Datos en Java**
    - 5.1 Arreglos unidimensionales
    - 5.2 Arreglos multidimensionales
    - 5.3 Iteración sobre arreglos
    - 5.4 Colecciones (List, Set, Map)
    - 5.5 Genéricos en colecciones
    - 5.6 Uso de ArrayList, HashMap, HashSet, TreeSet

- **Capítulo 6: Excepciones y Manejo de Errores en Java**
    - 6.1 Tipos de Excepciones
    - 6.2 Try, Catch, Finally
    - 6.3 Throws y Throw
    - 6.4 Creación de Excepciones Personalizadas
- **Capítulo 7: Entrada y Salida (I/O) en Java**
    - 7.1 Entrada estándar con Scanner
    - 7.2 Escritura y lectura de archivos
    - 7.3 Clases File, FileReader, FileWriter
    - 7.4 Serialización de objetos
- **Capítulo 8: Programación Funcional en Java**
    - 8.1 Interfaces Funcionales
    - 8.2 Expresiones Lambda
    - 8.3 La API Stream
    - 8.4 Operaciones Map, Filter, Reduce
    - 8.5 Optional y manejo de Null
- **Capítulo 9: Clases Utilitarias del API de Java**
    - 9.1 Clases utilitarias: Math, Arrays, Objects, Collections
    - 9.2 Date, LocalDate, LocalTime, LocalDateTime
    - 9.3 Formato de fechas y horas
    - 9.4 Temporizadores y tareas programadas
- **Capítulo 10: Programación Concurrente y Multihilo en Java**
    - 10.1 Conceptos de concurrencia
    - 10.2 Clase Thread y Runnable
    - 10.3 Sincronización
    - 10.4 Ejecutores y Callable
    - 10.5 Problemas comunes: race conditions, deadlocks
- **Capítulo 11: Desarrollo de Interfaces Gráficas con Swing**
    - 11.1 Introducción a Swing
    - 11.2 Ventanas, paneles, botones, etiquetas
    - 11.3 Campos de texto, áreas de texto, menús
    - 11.4 Layouts y organización de componentes
    - 11.5 Eventos y manejo de acciones
    - 11.6 Tablas y listas
- **Capítulo 12: Acceso a Bases de Datos con JDBC**
    - 12.1 Introducción a JDBC
    - 12.2 Conexión con bases de datos
    - 12.3 Sentencias SQL desde Java
    - 12.4 PreparedStatement y ResultSet
    - 12.5 Inserción, actualización y eliminación
    - 12.6 Manejo de transacciones
    - 12.7 Cierre y gestión de recursos

- **Capítulo 13: Programación Web con Java (Introducción)**
    - 13.1 Java EE y Java Jakarta
    - 13.2 Servlets y JSP
    - 13.3 Configuración de un servidor Tomcat
    - 13.4 Formularios HTML y envío a Servlets
    - 13.5 Conexión de Servlets a bases de datos
- **Capítulo 14: Introducción a Spring Framework**
    - 14.1 Inversión de Control (IoC) y Contenedores
    - 14.2 Spring Boot y creación de proyectos
    - 14.3 Inyección de dependencias
    - 14.4 Repositorios y controladores REST
    - 14.5 JPA y conexión a base de datos
    - 14.6 Arquitectura MVC en Spring
- **Capítulo 15: Pruebas y Depuración en Java**
    - 15.1 Introducción a JUnit
    - 15.2 Asserts y pruebas unitarias
    - 15.3 Mocking con Mockito
    - 15.4 Técnicas de depuración en IDEs
    - 15.5 Logs y seguimiento de errores
- **Capítulo 16: Buenas Prácticas y Patrones de Diseño en Java**
    - 16.1 Principios SOLID
    - 16.2 Refactorización de código
    - 16.3 Documentación con Javadoc
    - 16.4 Patrones de diseño: Singleton, Factory, Observer, MVC
- **Capítulo 17: Herramientas y Entornos de Desarrollo en Java**
    - 17.1 Uso de IDEs: IntelliJ IDEA, Eclipse, NetBeans
    - 17.2 Uso de Git y GitHub
    - 17.3 Automatización con Maven y Gradle
    - 17.4 Construcción de proyectos modulares
    - 17.5 Integración continua
- **Capítulo 18: Proyecto Final en Java**
    - 18.1 Diseño y planificación del proyecto
    - 18.2 Desarrollo en fases (modelo incremental)
    - 18.3 Pruebas y validación del sistema
    - 18.4 Documentación técnica y de usuario
    - 18.5 Presentación del proyecto
- **Capítulo 19: Recursos Adicionales**
    - 19.1 Libros recomendados
    - 19.2 Documentación oficial de Java
    - 19.3 Foros y comunidades
    - 19.4 Certificaciones Java
    - 19.5 Rutas de especialización: backend, Android, data science, enterprise

- **Ejercicios Prácticos**
    - Ejercicio 1 al Ejercicio 100

Epilogo

Capítulo 1: Introducción al lenguaje Java

1.1 Historia y evolución de Java

Java es un lenguaje de programación de propósito general, orientado a objetos, desarrollado por Sun Microsystems a principios de la década de 1990. El proyecto fue iniciado por James Gosling, Mike Sheridan y Patrick Naughton en 1991, como parte de un proyecto interno llamado "Green Project". El objetivo inicial era desarrollar un lenguaje para pequeños dispositivos electrónicos, como televisores interactivos.

En 1995, Sun Microsystems lanzó oficialmente Java al mercado. Su gran atractivo era la promesa de que los programas escritos en Java podían ejecutarse en cualquier dispositivo que tuviera una máquina virtual Java (JVM), gracias al lema "Write Once, Run Anywhere" (Escribe una vez, ejecútalo en cualquier parte).

A lo largo de los años, Java ha evolucionado significativamente:

- Java 1.0 (1996): primera versión estable.
- Java 2 (1998): introdujo las ediciones J2SE, J2EE y J2ME.
- Java 5 (2004): añadió características como genéricos, enumeraciones y anotaciones.
- Java 8 (2014): introdujo las expresiones lambda y Streams.
- Java 11 (2018): versión con soporte a largo plazo (LTS).
- Actualmente, Java es mantenido por Oracle Corporation y tiene una comunidad activa que colabora en su evolución continua.

Java se ha convertido en uno de los lenguajes más utilizados en el mundo, especialmente en el desarrollo de aplicaciones empresariales, móviles (Android), web y sistemas embebidos.

1.2 Características principales del lenguaje

Java tiene varias características que lo hacen atractivo y potente para desarrolladores:

- **Orientado a objetos**: Todo en Java se basa en clases y objetos, lo que permite una programación modular y reutilizable.
- **Independiente de la plataforma**: Gracias a la JVM, el mismo código fuente puede ejecutarse en diferentes sistemas operativos sin necesidad de modificaciones.
- **Sintaxis clara y familiar**: Su sintaxis está basada en C y C++, lo que lo hace accesible para programadores con experiencia en estos lenguajes.
- **Recolección de basura automática**: Java gestiona automáticamente la memoria, lo que reduce los errores relacionados con la gestión manual de memoria.
- **Seguridad**: Java proporciona un entorno de ejecución seguro, con mecanismos de control de acceso y validación de código.
- **Multihilo**: Java permite la ejecución de múltiples hilos de manera eficiente, lo que es útil para aplicaciones concurrentes.
- **Biblioteca estándar amplia**: Incluye una gran cantidad de clases y paquetes para realizar tareas comunes: entrada/salida, red, estructuras de datos, interfaces gráficas, etc.

- **Desarrollo distribuido**: Java facilita la creación de aplicaciones que se comunican a través de redes mediante sockets, RMI, etc.
- **Documentación integrada**: Con herramientas como Javadoc, se puede generar documentación de manera automática.

1.3 Tipos de aplicaciones Java

Java es un lenguaje versátil que permite desarrollar varios tipos de aplicaciones:

- **Aplicaciones de escritorio**: Programas con interfaz gráfica para PC, usando herramientas como AWT, Swing o JavaFX.
- **Aplicaciones web**: Usando tecnologías como Servlets, JSP, Spring o frameworks como Struts.
- **Aplicaciones móviles**: Principalmente para Android, ya que el sistema operativo está basado en Java.
- **Aplicaciones empresariales**: Utilizando plataformas como Java EE o Jakarta EE para sistemas complejos de negocios.
- **Aplicaciones embebidas**: Java puede ejecutarse en dispositivos con recursos limitados, como lectores de tarjetas o electrodomésticos.
- **Aplicaciones científicas**: Java también es usado para simulaciones, análisis y procesamiento numérico, gracias a su estabilidad y escalabilidad.

1.4 Instalación del JDK y configuración del entorno

Para programar en Java, se necesita instalar el JDK (Java Development Kit), que incluye:

- El compilador `javac`.
- La Máquina Virtual Java (JVM).
- Herramientas de desarrollo (Javadoc, debugger, etc.).
- Bibliotecas estándar.

Pasos para instalar el JDK:

1. Ir al sitio oficial de Oracle (https://www.oracle.com/java/technologies/javase-downloads.html).
2. Descargar la última versión del JDK para tu sistema operativo.
3. Ejecutar el instalador y seguir los pasos de instalación.
4. Configurar las variables de entorno:
    - **JAVA_HOME**: debe apuntar al directorio donde está instalado el JDK.
      Ejemplo en Windows:
      `JAVA_HOME=C:\Program Files\Java\jdk-21`
    - **PATH**: añadir la carpeta `bin` del JDK al PATH del sistema.
      Ejemplo:
      `C:\Program Files\Java\jdk-21\bin`

5. Verificar la instalación desde la línea de comandos con:

   ```
   java -version
   javac -version
   ```

1.5 Primer programa en Java

A continuación, se presenta un ejemplo clásico de "Hola mundo" en Java:

```
public class HolaMundo {
    public static void main(String[] args) {
        System.out.println("¡Hola, mundo!");
    }
}
```

Explicación del código:

- `public class HolaMundo`: se declara una clase pública llamada `HolaMundo`. En Java, todo el código debe estar dentro de una clase.
- `public static void main(String[] args)`: es el método principal. Es el punto de entrada de una aplicación Java.
- `System.out.println("¡Hola, mundo!");`: imprime el texto en la consola.

1.6 Uso de la línea de comandos y compilación

Java permite compilar y ejecutar programas desde la línea de comandos:

Pasos para compilar y ejecutar:

1. Crear un archivo llamado `HolaMundo.java` con el código fuente.
2. Abrir una terminal o símbolo del sistema, ubicarse en el directorio donde está el archivo.
3. Compilar el programa con el comando:

   ```
   javac HolaMundo.java
   ```

   Esto generará un archivo `HolaMundo.class`, que contiene el bytecode.
4. Ejecutar el programa con:

   ```
   java HolaMundo
   ```

   Nota: no se incluye la extensión `.class` al ejecutar.

Errores comunes:

- No poner el mismo nombre de archivo que el nombre de la clase pública.
- Olvidar configurar correctamente el PATH.
- Escribir mal los comandos (`java` vs `javac`).

Conclusión del capítulo:

En este capítulo hemos introducido el lenguaje Java, repasando su historia, sus características más importantes, los tipos de aplicaciones que se pueden desarrollar y cómo instalar el entorno de desarrollo. También aprendimos a compilar y ejecutar un programa básico desde la línea de comandos. Estos conocimientos básicos sientan las bases para avanzar hacia conceptos más complejos en el desarrollo con Java.

Capítulo 2: Fundamentos del lenguaje Java

2.1 Tipos de datos primitivos

Java es un lenguaje fuertemente tipado, lo que significa que cada variable debe declararse con un tipo específico. Existen ocho tipos de datos primitivos en Java:

- `byte`: Entero de 8 bits. Rango: -128 a 127.
- `short`: Entero de 16 bits. Rango: -32,768 a 32,767.
- `int`: Entero de 32 bits. Rango: -2^31 a 2^31-1.
- `long`: Entero de 64 bits. Se usa para números grandes. Se indica con una `L` al final.
- `float`: Número en punto flotante de 32 bits. Se indica con una `f` al final.
- `double`: Número en punto flotante de 64 bits (más preciso que `float`).
- `char`: Representa un solo carácter Unicode (16 bits).
- `boolean`: Representa verdadero (`true`) o falso (`false`).

Ejemplos:

```
int edad = 25;
float precio = 19.99f;
char inicial = 'A';
boolean activo = true;
```

2.2 Variables y constantes

Una variable es un espacio en memoria para almacenar un valor que puede cambiar durante la ejecución del programa. Se declara indicando el tipo seguido del nombre:

```
int contador;
contador = 10;
```

También se puede inicializar al declarar:

```
int numero = 5;
```

Las constantes son variables cuyo valor no puede cambiar una vez asignado. Se declaran usando `final`:

```
final double PI = 3.1416;
```

Por convención, los nombres de constantes se escriben en mayúsculas.

2.3 Operadores aritméticos, lógicos y relacionales

- Aritméticos:
  - `+` Suma
  - `-` Resta
  - `*` Multiplicación

    - `/` División
    - `%` Módulo (resto de la división)

Ejemplo:

```
int resultado = 10 + 5;  // 15
```

- Relacionales:
    - == Igual a
    - `!=` Distinto de
    - > Mayor que
    - < Menor que
    - >= Mayor o igual que
    - <= Menor o igual que

Ejemplo:

```
boolean mayor = 5 > 3;  // true
```

- Lógicos:
    - && AND lógico
    - `||` OR lógico
    - `!` NOT lógico

Ejemplo:

```
boolean resultado = (5 > 3) && (4 < 10);  // true
```

2.4 Conversión de tipos

Java permite conversiones automáticas (promociones) entre tipos compatibles, pero algunas requieren casting explícito.

Conversión implícita:

```
int x = 10;
double y = x;  // conversión automática de int a double
```

Conversión explícita:

```
double a = 9.7;
int b = (int) a;  // b será 9
```

Precaución: la conversión explícita puede llevar a pérdida de datos.

También se pueden usar métodos de las clases wrapper:

```
String texto = "123";
int num = Integer.parseInt(texto);  // conversión de String a int
```

2.5 Estructuras de control (if, switch, while, for, do-while)

- `if` / `else if` / `else`:

```
int edad = 20;
if (edad >= 18) {
    System.out.println("Mayor de edad");
} else {
    System.out.println("Menor de edad");
}
```

- `switch`:

```
int dia = 3;
switch (dia) {
    case 1:
        System.out.println("Lunes");
        break;
    case 2:
        System.out.println("Martes");
        break;
    default:
        System.out.println("Otro día");
}
```

- `while`:

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

- `do-while`:

```
int j = 0;
do {
    System.out.println(j);
    j++;
} while (j < 5);
```

- `for`:

```
for (int k = 0; k < 5; k++) {
    System.out.println(k);
}
```

2.6 Expresiones booleanas y operadores ternarios

Las expresiones booleanas evalúan a `true` o `false`. Son esenciales en estructuras de control.

Ejemplo:

```
boolean esPar = (numero % 2 == 0);
```

Operador ternario:

```
int edad = 20;
String mensaje = (edad >= 18) ? "Adulto" : "Menor";
```

El operador ternario es una forma abreviada del `if`.

2.7 Sentencias break, continue y return

- `break`: Sale del bucle o estructura `switch`.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
    System.out.println(i);
}
```

- `continue`: Omite la iteración actual y pasa a la siguiente.

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) continue;
    System.out.println(i);  // imprimirá solo los impares
}
```

- `return`: Finaliza la ejecución de un método y puede devolver un valor.

```
public int suma(int a, int b) {
    return a + b;
}
```

En métodos `void`, `return` se usa solo para salir anticipadamente:

```
public void mostrar() {
    if (false) return;
    System.out.println("Visible");
}
```

}

CAPÍTULO 3: PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA

3.1 CLASES Y OBJETOS

En Java, la programación orientada a objetos (POO) permite modelar el mundo real a través de clases y objetos. Una clase es un plano o plantilla que define la estructura y el comportamiento de los objetos. Un objeto es una instancia de una clase, es decir, una entidad concreta que posee atributos (estado) y métodos (comportamiento).

Ejemplo:

class Persona {
String nombre;
int edad;

```
void saludar() {
    System.out.println("Hola, mi nombre es " + nombre);
}
```

}

public class Principal {
public static void main(String[] args) {
Persona p1 = new Persona();
p1.nombre = "Ana";
p1.edad = 30;
p1.saludar();
}
}

En este ejemplo, "Persona" es una clase con dos atributos y un método. Luego se crea un objeto "p1" de tipo Persona.

3.2 CONSTRUCTORES

Un constructor es un método especial que se ejecuta al crear un objeto. Se usa para inicializar los atributos. Tiene el mismo nombre que la clase y no tiene tipo de retorno.

Ejemplo:

class Persona {
String nombre;
int edad;

```
Persona(String nom, int ed) {
    nombre = nom;
    edad = ed;
}

void mostrar() {
    System.out.println(nombre + " tiene " + edad + " años.");
}
```

}

public class Principal {
public static void main(String[] args) {
Persona p = new Persona("Luis", 25);
p.mostrar();
}
}

3.3 ATRIBUTOS Y MÉTODOS

Los atributos son variables dentro de una clase que representan el estado de los objetos. Los métodos son funciones que definen el comportamiento.

Ejemplo:

class Coche {
String marca;
int velocidad;

```
void acelerar() {
    velocidad += 10;
}

void mostrarVelocidad() {
    System.out.println("Velocidad actual: " + velocidad);
}
```

}

3.4 ENCAPSULAMIENTO

El encapsulamiento consiste en ocultar los detalles internos de una clase y exponer sólo lo necesario a través de métodos públicos. Se logra usando modificadores de acceso y métodos getters y setters.

Ejemplo:

class CuentaBancaria {
private double saldo;

```
public void depositar(double cantidad) {
    if (cantidad > 0) saldo += cantidad;
}

public double getSaldo() {
    return saldo;
}
```

}

3.5 MODIFICADORES DE ACCESO

Java define modificadores que controlan la visibilidad de los elementos de una clase:

- public: accesible desde cualquier clase.
- private: accesible solo dentro de la clase.
- protected: accesible en la clase, en el paquete y en subclases.
- sin modificador (default): accesible solo dentro del mismo paquete.

Ejemplo:

```
class Ejemplo {
public int publico;
private int privado;
protected int protegido;
int paquete;
}
```

3.6 HERENCIA

La herencia permite que una clase herede atributos y métodos de otra clase. Se usa la palabra clave "extends".

Ejemplo:

```
class Animal {
void comer() {
System.out.println("El animal come");
}
}

class Perro extends Animal {
void ladrar() {
System.out.println("El perro ladra");
}
}

public class Principal {
public static void main(String[] args) {
Perro p = new Perro();
p.comer();
p.ladrar();
}
}
```

3.7 POLIMORFISMO

El polimorfismo permite que un mismo método se comporte de manera diferente según el objeto que lo invoque.

Ejemplo:

```
class Animal {
void hacerSonido() {
System.out.println("Sonido genérico");
}
}

class Gato extends Animal {
void hacerSonido() {
System.out.println("Miau");
}
}

class Vaca extends Animal {
void hacerSonido() {
System.out.println("Muu");
}
}

public class Principal {
public static void main(String[] args) {
Animal a1 = new Gato();
Animal a2 = new Vaca();
a1.hacerSonido();
a2.hacerSonido();
}
}
```

3.8 SOBRECARGA DE MÉTODOS Y CONSTRUCTORES

La sobrecarga permite definir varios métodos o constructores con el mismo nombre pero diferente lista de parámetros.

Ejemplo:

```
class Calculadora {
int sumar(int a, int b) {
return a + b;
}

double sumar(double a, double b) {
    return a + b;
}

}
```

Ejemplo de constructores sobrecargados:

class Persona {
String nombre;
int edad;

```
Persona() {
    nombre = "Sin nombre";
    edad = 0;
}

Persona(String nom) {
    nombre = nom;
    edad = 0;
}

Persona(String nom, int ed) {
    nombre = nom;
    edad = ed;
}
```

}

3.9 CLASES ABSTRACTAS

Una clase abstracta no se puede instanciar y puede contener métodos abstractos (sin cuerpo). Se usa como base para otras clases.

Ejemplo:

abstract class Figura {
abstract double area();
}

class Circulo extends Figura {
double radio;

```
Circulo(double r) {
    radio = r;
}

double area() {
    return Math.PI * radio * radio;
}
```

}

3.10 INTERFACES

Una interfaz define un contrato que las clases deben implementar. Todos los métodos de una interfaz son públicos y abstractos por defecto.

Ejemplo:

```
interface Vehiculo {
void arrancar();
void frenar();
}

class Moto implements Vehiculo {
public void arrancar() {
System.out.println("Moto arrancando");
}

public void frenar() {
    System.out.println("Moto frenando");
}

}
```

3.11 USO DE THIS Y SUPER

- this: se refiere al objeto actual.
- super: se refiere a la superclase.

Ejemplo de this:

```
class Persona {
String nombre;

Persona(String nombre) {
    this.nombre = nombre;
}

}
```

Ejemplo de super:

```
class Animal {
void hacerSonido() {
System.out.println("Sonido genérico");
}
}

class Perro extends Animal {
void hacerSonido() {
super.hacerSonido();
System.out.println("Guau");
}
}
```

3.12 ENUMERACIONES

Una enumeración es un tipo que define un conjunto fijo de constantes.

Ejemplo:

```
enum Dia {
LUNES, MARTES, MIERCOLES, JUEVES, VIERNES
}
```

```
public class Principal {
public static void main(String[] args) {
Dia d = Dia.LUNES;
System.out.println("Hoy es " + d);
}
}
```

CAPÍTULO 4: MANEJO DE CADENAS DE TEXTO EN JAVA

4.1 LA CLASE STRING

En Java, las cadenas de texto son representadas mediante la clase String, que forma parte del paquete java.lang. Esta clase permite trabajar con secuencias de caracteres de forma sencilla, ya que provee una gran cantidad de métodos útiles para manipular, comparar y transformar cadenas. Las instancias de String son inmutables, lo que significa que una vez creada una cadena, su contenido no puede cambiar. Cualquier operación que modifique una cadena en realidad devuelve una nueva instancia de String.

Creación de cadenas:

```
String cadena1 = "Hola mundo";
String cadena2 = new String("Hola mundo");
```

Ambas formas son válidas, pero la primera es más eficiente porque utiliza el pool de cadenas que Java mantiene internamente para optimizar el uso de memoria.

Longitud de una cadena:

```
int longitud = cadena1.length();
```

Acceso a un carácter específico:

```
char letra = cadena1.charAt(0); // Devuelve 'H'
```

Subcadenas:

```
String sub = cadena1.substring(5); // Devuelve "mundo"
String sub2 = cadena1.substring(0, 4); // Devuelve "Hola"
```

4.2 MÉTODOS ÚTILES DE STRING

La clase String incluye una gran variedad de métodos para manipular y consultar cadenas. A continuación, se describen algunos de los más comunes y útiles.

equals y equalsIgnoreCase:

Permiten comparar el contenido de dos cadenas.

```
String a = "Hola";
String b = "hola";
```

```
boolean igual = a.equals(b); // false
boolean igualIgnorarMayusculas = a.equalsIgnoreCase(b); // true
```

compareTo:

Compara dos cadenas lexicográficamente.

```
String a = "manzana";
String b = "naranja";
```

```
int resultado = a.compareTo(b); // resultado negativo porque "manzana" va antes que "naranja"
```

contains:

Verifica si una cadena contiene una subcadena.

```
String texto = "Bienvenido al mundo Java";
boolean contiene = texto.contains("Java"); // true
```

startsWith y endsWith:

Verifican si una cadena empieza o termina con otra.

```
boolean empieza = texto.startsWith("Bienvenido"); // true
boolean termina = texto.endsWith("Java"); // true
```

indexOf y lastIndexOf:

Devuelven la posición de una subcadena o carácter.

```
int posicion = texto.indexOf("mundo"); // 15
int ultimaOcurrencia = texto.lastIndexOf("o"); // 20
```

toUpperCase y toLowerCase:

Transforman la cadena a mayúsculas o minúsculas.

```
String mayusculas = texto.toUpperCase();
String minusculas = texto.toLowerCase();
```

trim:

Elimina espacios en blanco al principio y al final.

```
String mensaje = " Hola ";
String limpio = mensaje.trim(); // "Hola"
```

replace:

Reemplaza caracteres o subcadenas.

```
String texto = "perro";
String reemplazado = texto.replace("r", "l"); // "pelro"
```

split:

Divide la cadena en partes usando un delimitador.

```
String frase = "uno,dos,tres";
String[] partes = frase.split(","); // ["uno", "dos", "tres"]
```

4.3 STRINGBUILDER Y STRINGBUFFER

La clase String es inmutable, por lo tanto, si se necesita modificar cadenas de forma frecuente y eficiente, se recomienda utilizar StringBuilder o StringBuffer.

StringBuilder es una clase mutable que permite modificar el contenido de una cadena sin crear nuevos objetos. StringBuffer ofrece lo mismo, pero es sincronizado, lo cual lo hace adecuado en contextos con múltiples hilos.

Ejemplo de uso de StringBuilder:

```
StringBuilder sb = new StringBuilder();
sb.append("Hola");
sb.append(" ");
sb.append("mundo");
String resultado = sb.toString(); // "Hola mundo"
```

Principales métodos de StringBuilder y StringBuffer:

- append(String s): añade una cadena al final
- insert(int offset, String s): inserta una cadena en la posición indicada
- delete(int start, int end): elimina los caracteres entre las posiciones start y end
- reverse(): invierte el contenido
- replace(int start, int end, String s): reemplaza una parte de la cadena
- toString(): convierte el objeto en un String

Ejemplo:

```
StringBuilder sb = new StringBuilder("Hola");
sb.insert(4, " mundo"); // "Hola mundo"
sb.delete(0, 5); // "mundo"
sb.reverse(); // "odnum"
String resultado = sb.toString();
```

Diferencias entre String, StringBuilder y StringBuffer:

- String: inmutable, menos eficiente para modificaciones repetidas.
- StringBuilder: mutable, no sincronizado, más rápido en entornos de un solo hilo.
- StringBuffer: mutable, sincronizado, más lento pero seguro en entornos multihilo.

4.4 COMPARACIÓN Y MANIPULACIÓN DE CADENAS

Comparación de cadenas:

Existen varias formas de comparar cadenas en Java, según lo que se desea comparar.

1. Comparación por contenido (equals y equalsIgnoreCase)

```
String a = "Java";
String b = "Java";
String c = new String("Java");
```

```
a == b; // true (porque ambos apuntan al mismo objeto del pool)
a == c; // false (son objetos diferentes)
a.equals(c); // true (contenido igual)
```

2. Comparación lexicográfica (compareTo)

compareTo devuelve:

- 0 si las cadenas son iguales

- Un número negativo si la primera cadena es menor
- Un número positivo si la primera cadena es mayor

3. Uso de región:

Se puede comparar partes de una cadena con regionMatches.

```
String a = "Hola mundo";
boolean region = a.regionMatches(5, "mundo", 0, 5); // true
```

Manipulación avanzada:

1. Eliminación de espacios:

```
String texto = " ejemplo ";
String limpio = texto.trim(); // "ejemplo"
```

2. Reemplazo múltiple:

```
String texto = "casa azul azul";
texto = texto.replaceAll("azul", "roja"); // "casa roja roja"
```

3. Extracción de palabras:

```
String oracion = "Java es un lenguaje";
String[] palabras = oracion.split(" ");

for (String palabra : palabras) {
System.out.println(palabra);
}
```

4. Validación de cadenas:

```
String email = "usuario@correo.com";
boolean valido = email.contains("@") && email.endsWith(".com");
```

5. Concatenación eficiente:

En lugar de usar + repetidamente, es más eficiente usar StringBuilder en bucles:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10; i++) {
sb.append(i).append(" ");
}
String resultado = sb.toString();
```

CONCLUSIÓN DEL CAPÍTULO

El manejo de cadenas de texto en Java es fundamental en cualquier tipo de aplicación. La clase String proporciona una base sólida para representar texto, mientras que StringBuilder y StringBuffer permiten optimizar el rendimiento cuando se requiere modificar cadenas repetidamente. Entender y dominar los métodos de comparación y manipulación de cadenas permite escribir código más limpio, eficiente y profesional.

Capítulo 5: Arreglos y Estructuras de Datos en Java

5.1 Arreglos unidimensionales

Un arreglo unidimensional en Java es una estructura de datos que almacena múltiples elementos del mismo tipo en una secuencia contigua de memoria. Los arreglos se utilizan cuando se conoce de antemano el número de elementos que se desean almacenar.

Declaración y creación de un arreglo:

```
int[] numeros; // Declaración
numeros = new int[5]; // Creación
```

También se puede declarar y crear en una sola línea:

```
int[] numeros = new int[5];
```

Inicialización de valores:

```
numeros[0] = 10;
numeros[1] = 20;
numeros[2] = 30;
numeros[3] = 40;
numeros[4] = 50;
```

O usando una sintaxis abreviada:

```
int[] numeros = {10, 20, 30, 40, 50};
```

Acceso a elementos:

```
System.out.println(numeros[2]); // Muestra 30
```

Longitud del arreglo:

```
System.out.println(numeros.length); // Muestra 5
```

Ventajas:

- Acceso rápido a los elementos mediante índices.
- Simplicidad.

Desventajas:

- Tamaño fijo una vez declarado.
- No permite eliminar o insertar elementos dinámicamente.

5.2 Arreglos multidimensionales

Un arreglo multidimensional es un arreglo de arreglos. El más común es el arreglo bidimensional, que se utiliza para representar matrices.

Declaración y creación:

```
int[][] matriz = new int[3][4]; // Matriz de 3 filas y 4 columnas
```

Inicialización:

```
int[][] matriz = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

Acceso a elementos:

```
System.out.println(matriz[1][2]); // Muestra 7
```

Recorrido de una matriz:

```
for (int i = 0; i < matriz.length; i++) {
    for (int j = 0; j < matriz[i].length; j++) {
        System.out.print(matriz[i][j] + " ");
    }
    System.out.println();
}
```

Los arreglos multidimensionales también pueden tener dimensiones irregulares:

```
int[][] arregloIrregular = new int[3][];
arregloIrregular[0] = new int[2];
arregloIrregular[1] = new int[4];
arregloIrregular[2] = new int[3];
```

5.3 Iteración sobre arreglos

Hay varias formas de iterar sobre arreglos en Java:

Bucle `for` tradicional:

```
int[] numeros = {1, 2, 3, 4, 5};

for (int i = 0; i < numeros.length; i++) {
    System.out.println(numeros[i]);
}
```

Bucle `for-each`:

```
for (int numero : numeros) {
    System.out.println(numero);
}
```

Este bucle es más limpio, pero no permite modificar los elementos directamente ni conocer su índice.

5.4 Colecciones (List, Set, Map)

Las colecciones son estructuras de datos dinámicas proporcionadas por Java para almacenar, manipular y recuperar grupos de objetos.

List: Una colección ordenada que puede contener elementos duplicados.

Set: Una colección que no permite duplicados.

Map: Una colección de pares clave-valor.

5.5 Genéricos en colecciones

Las colecciones en Java son genéricas, lo que significa que puedes especificar el tipo de elementos que almacenan.

Ejemplo con List:

```
List<String> nombres = new ArrayList<>();
nombres.add("Ana");
nombres.add("Luis");
```

Esto garantiza que sólo se puedan añadir elementos de tipo `String`.

Ventajas de los genéricos:

- Seguridad de tipos en tiempo de compilación.
- No es necesario hacer casting al recuperar elementos.
- Código más legible y mantenible.

5.6 Uso de ArrayList, HashMap, HashSet, TreeSet

ArrayList:
Una lista dinámica basada en arreglos.

```
import java.util.ArrayList;

ArrayList<String> frutas = new ArrayList<>();
frutas.add("Manzana");
frutas.add("Banana");
frutas.add("Cereza");

System.out.println(frutas.get(1)); // Banana
```

HashMap:
Estructura de pares clave-valor. No garantiza orden.

```
import java.util.HashMap;

HashMap<String, Integer> edades = new HashMap<>();
edades.put("Ana", 25);
edades.put("Luis", 30);

System.out.println(edades.get("Ana")); // 25
```

HashSet:
Colección que no permite duplicados y no garantiza orden.

```
import java.util.HashSet;

HashSet<String> colores = new HashSet<>();
colores.add("Rojo");
colores.add("Verde");
colores.add("Rojo"); // No se añade otra vez

System.out.println(colores.size()); // 2
```

TreeSet:
Colección sin duplicados que mantiene los elementos ordenados.

```
import java.util.TreeSet;

TreeSet<Integer> numeros = new TreeSet<>();
numeros.add(5);
numeros.add(1);
numeros.add(3);

System.out.println(numeros); // [1, 3, 5]
```

Resumen del capítulo:

- Los arreglos unidimensionales y multidimensionales son estructuras fundamentales para almacenar datos homogéneos.
- Las colecciones como `List`, `Set` y `Map` permiten estructuras de datos más flexibles y potentes.
- Los genéricos permiten trabajar con colecciones de forma segura y clara.
- Las implementaciones comunes como `ArrayList`, `HashMap`, `HashSet` y `TreeSet` proporcionan herramientas listas para usar en distintas situaciones.

Capítulo 6: Excepciones y Manejo de Errores en Java

6.1 Tipos de Excepciones

En Java, las excepciones son eventos que alteran el flujo normal de ejecución de un programa. Ocurren cuando se presenta una condición anómala, como intentar dividir entre cero o acceder a un índice fuera de los límites de un array.

Java clasifica las excepciones en tres grandes tipos:

1. **Excepciones comprobadas (Checked Exceptions)**
   Estas excepciones son verificadas en tiempo de compilación. El compilador obliga al programador a manejar estas excepciones mediante bloques `try-catch` o utilizando la cláusula `throws`.
   Ejemplos:

- `IOException`
- `SQLException`

```
import java.io.FileReader;
import java.io.IOException;

public class EjemploChecked {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("archivo.txt");
        } catch (IOException e) {
            System.out.println("Archivo no encontrado.");
        }
    }
}
```

2. **Excepciones no comprobadas (Unchecked Exceptions)**
   Estas excepciones derivan de `RuntimeException` y no requieren ser manejadas en tiempo de compilación.
   Ejemplos:

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`

```
public class EjemploUnchecked {
    public static void main(String[] args) {
        int[] numeros = {1, 2, 3};
        System.out.println(numeros[5]); // Lanza ArrayIndexOutOfBoundsException
    }
}
```

3. **Errores (Errors)**
   Son condiciones graves del sistema que generalmente no deberían ser capturadas por el programa.
   Ejemplos:

- `OutOfMemoryError`
- `StackOverflowError`

```
public class EjemploError {
    public static void recursividadInfinita() {
        recursividadInfinita(); // StackOverflowError
    }

    public static void main(String[] args) {
        recursividadInfinita();
    }
}
```

6.2 Try, Catch, Finally

Java proporciona bloques `try-catch-finally` para manejar las excepciones en tiempo de ejecución. La estructura básica es:

```
try {
    // Código que puede lanzar una excepción
} catch (TipoDeExcepcion e) {
    // Código para manejar la excepción
} finally {
    // Código que se ejecuta siempre
}
```

- El bloque `try` contiene el código susceptible de generar una excepción.
- El bloque `catch` captura y maneja la excepción.
- El bloque `finally` se ejecuta siempre, ocurra o no una excepción. Se usa para liberar recursos como archivos o conexiones.

Ejemplo con finally:

```
public class EjemploFinally {
    public static void main(String[] args) {
        try {
            int resultado = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("No se puede dividir por cero.");
        } finally {
            System.out.println("Este bloque siempre se ejecuta.");
        }
    }
}
```

Múltiples bloques catch:

```
try {
    int[] arr = new int[5];
    arr[10] = 4;
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Índice fuera de rango.");
} catch (Exception e) {
    System.out.println("Ocurrió una excepción.");
}
```

6.3 Throws y Throw

throw: se utiliza para lanzar explícitamente una excepción desde un método o bloque de código.

throws: se utiliza en la declaración de métodos para indicar que un método puede lanzar excepciones.

Ejemplo con throw:

```
public class EjemploThrow {
    public static void verificarEdad(int edad) {
        if (edad < 18) {
            throw new ArithmeticException("Debe ser mayor de edad.");
        }
        System.out.println("Acceso permitido.");
    }

    public static void main(String[] args) {
        verificarEdad(15);
    }
}
```

Ejemplo con throws:

```
import java.io.IOException;

public class EjemploThrows {
    public static void leerArchivo() throws IOException {
        throw new IOException("Error al leer archivo.");
    }

    public static void main(String[] args) {
        try {
            leerArchivo();
        } catch (IOException e) {
            System.out.println("Manejo de IOException.");
        }
    }
}
```

Reglas importantes:

- Un método puede declarar múltiples excepciones separadas por comas.
- Si una excepción es comprobada, debe ser manejada o declarada con throws.

6.4 Creación de Excepciones Personalizadas

Java permite crear nuestras propias excepciones definiendo una clase que extienda de `Exception` (checked) o `RuntimeException` (unchecked).

Ejemplo de excepción personalizada checked:

```
class MiExcepcion extends Exception {
    public MiExcepcion(String mensaje) {
        super(mensaje);
    }
}

public class PruebaExcepcion {
```

```
    public static void metodoPeligroso(boolean error) throws MiExcepcion {
        if (error) {
            throw new MiExcepcion("Se produjo un error personalizado.");
        }
        System.out.println("Método ejecutado correctamente.");
    }

    public static void main(String[] args) {
        try {
            metodoPeligroso(true);
        } catch (MiExcepcion e) {
            System.out.println("Capturada: " + e.getMessage());
        }
    }
}
```

Ejemplo de excepción personalizada unchecked:

```
class MiExcepcionRuntime extends RuntimeException {
    public MiExcepcionRuntime(String mensaje) {
        super(mensaje);
    }
}

public class EjemploUncheckedPersonalizada {
    public static void main(String[] args) {
        throw new MiExcepcionRuntime("Error inesperado personalizado.");
    }
}
```

Resumen del Capítulo:

- Java maneja errores mediante un sistema robusto de excepciones.
- Las excepciones pueden ser comprobadas, no comprobadas o errores graves.
- Se manejan con `try`, `catch`, `finally`.
- `throw` se usa para lanzar excepciones manualmente; `throws` para declararlas en métodos.
- Las excepciones personalizadas permiten definir errores específicos para la lógica de la aplicación.

CAPÍTULO 8 - PROGRAMACIÓN FUNCIONAL EN JAVA

8.1 INTERFACES FUNCIONALES

Una interfaz funcional en Java es una interfaz que contiene un único método abstracto. Estas interfaces permiten representar funciones como objetos, y son la base para utilizar expresiones lambda.

Desde Java 8, se introdujo la anotación @FunctionalInterface para indicar explícitamente que una interfaz es funcional. Esto no es obligatorio, pero ayuda a documentar la intención del programador y hace que el compilador verifique que solo haya un método abstracto.

Ejemplo:

```
@FunctionalInterface
interface Operacion {
int aplicar(int a, int b);
}
```

Las interfaces funcionales más comunes en Java (disponibles en el paquete java.util.function) son:

- Predicate: representa una función que toma un argumento y devuelve un booleano.
- Function<T, R>: representa una función que toma un argumento de tipo T y devuelve un resultado de tipo R.
- Consumer: representa una operación que toma un argumento de tipo T y no devuelve nada.
- Supplier: representa una función que no toma argumentos y devuelve un valor de tipo T.
- UnaryOperator: toma un argumento y devuelve un resultado del mismo tipo.
- BinaryOperator: toma dos argumentos del mismo tipo y devuelve uno del mismo tipo.

Estas interfaces se utilizan ampliamente en Streams, lambdas y otras construcciones funcionales.

8.2 EXPRESIONES LAMBDA

Las expresiones lambda son una forma concisa de representar una instancia de una interfaz funcional. Permiten escribir funciones como argumentos de métodos, evitando la necesidad de clases anónimas verbosas.

Sintaxis básica:

```
(parámetros) -> expresión
```

o

```
(parámetros) -> { cuerpo de la función }
```

Ejemplos:

```
Operacion suma = (a, b) -> a + b;
Operacion multiplicacion = (a, b) -> {
int resultado = a * b;
return resultado;
};
```

También se pueden usar lambdas con las interfaces estándar:

```
Predicate esVacio = s -> s.isEmpty();
Function<String, Integer> longitud = s -> s.length();
Consumer imprimir = s -> System.out.println(s);
```

8.3 LA API STREAM

La API Stream de Java permite procesar colecciones de forma funcional, encadenando operaciones como map, filter, reduce, etc. Un Stream no almacena datos, sino que representa una secuencia de operaciones que se pueden aplicar a elementos.

Se puede crear un Stream desde una colección usando el método stream():

```
List nombres = Arrays.asList("Ana", "Luis", "Pedro");
Stream flujo = nombres.stream();
```

Las operaciones en un Stream pueden ser intermedias (devuelven otro Stream) o terminales (devuelven un resultado y cierran el Stream). Las intermedias son lazy, es decir, no se ejecutan hasta que se invoca una operación terminal.

Ejemplo de uso:

```
List nombres = Arrays.asList("Ana", "Luis", "Pedro", "Laura");

nombres.stream()
.filter(n -> n.length() > 3)
.map(String::toUpperCase)
.forEach(System.out::println);
```

Esto imprime los nombres con más de 3 letras en mayúsculas.

8.4 OPERACIONES MAP, FILTER, REDUCE

Las operaciones más utilizadas en Streams son:

- map: transforma cada elemento del Stream usando una función.
- filter: selecciona solo los elementos que cumplen una condición.
- reduce: acumula los elementos del Stream en un solo resultado.

Ejemplos:

map:

```
List palabras = Arrays.asList("uno", "dos", "tres");
List longitudes = palabras.stream()
.map(String::length)
.collect(Collectors.toList());
```

filter:

```
List numeros = Arrays.asList(1, 2, 3, 4, 5, 6);
List pares = numeros.stream()
.filter(n -> n % 2 == 0)
.collect(Collectors.toList());
```

reduce:

```
List nums = Arrays.asList(1, 2, 3, 4);
int suma = nums.stream()
.reduce(0, (a, b) -> a + b);
```

También se puede usar reduce con Optional:

```
Optional resultado = nums.stream().reduce((a, b) -> a * b);
```

En este caso, el resultado puede estar presente o no si el Stream está vacío.

8.5 OPTIONAL Y MANEJO DE NULL

Optional es una clase contenedor introducida en Java 8 para representar valores que pueden estar presentes o ausentes, ayudando a evitar errores de null pointer.

Optional puede contener un valor no nulo, o estar vacío. Se crea con:

```
Optional posible = Optional.of("dato");
Optional vacio = Optional.empty();
Optional desdeMetodo = Optional.ofNullable(obtenerValor());
```

Se puede trabajar con Optional de forma funcional:

```
String resultado = posible
.map(String::toUpperCase)
.orElse("valor por defecto");
```

Optional también permite filtrar y encadenar operaciones:

```
Optional resultado = posible
.filter(s -> s.length() > 3)
.map(s -> s + "!!!");
```

Evita usar get() directamente, porque puede lanzar NoSuchElementException si no hay valor. Es preferible usar orElse, orElseGet o ifPresent.

Ejemplo práctico:

```
public Optional obtenerUsuario(String id) {
if (id.equals("123")) {
return Optional.of("Juan");
} else {
return Optional.empty();
}
}
```

```
obtenerUsuario("123")
.ifPresent(nombre -> System.out.println("Hola " + nombre));
```

CAPÍTULO 7: ENTRADA Y SALIDA (I/O) EN JAVA

La entrada/salida (I/O) en Java es una parte fundamental del lenguaje que permite la interacción del programa con el mundo exterior. Esto puede incluir la lectura de datos del teclado, la escritura o lectura de archivos en disco, o incluso la comunicación en red. En este capítulo exploraremos las operaciones básicas de entrada y salida en Java.

7.1 ENTRADA ESTÁNDAR CON SCANNER

La clase Scanner, introducida en Java 5, se utiliza comúnmente para obtener entrada desde diversas fuentes como el teclado, archivos, cadenas, etc. Para leer desde el teclado (entrada estándar), se utiliza Scanner junto con System.in.

EJEMPLO DE USO BÁSICO DE SCANNER:

```
import java.util.Scanner;

public class EntradaScanner {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Introduce tu nombre: ");
String nombre = sc.nextLine();
System.out.print("Introduce tu edad: ");
int edad = sc.nextInt();
System.out.println("Hola " + nombre + ", tienes " + edad + " años.");
sc.close();
}
}
```

Scanner tiene métodos como:

- nextLine() -> lee una línea completa
- next() -> lee una palabra
- nextInt(), nextDouble(), nextBoolean(), etc. -> leen tipos primitivos

Es importante cerrar el Scanner con close() para liberar recursos.

7.2 ESCRITURA Y LECTURA DE ARCHIVOS

Java proporciona varias clases en el paquete java.io para manipular archivos. A continuación se muestran dos operaciones básicas: escribir y leer archivos de texto.

ESCRIBIR EN UN ARCHIVO DE TEXTO:

```
import java.io.FileWriter;
import java.io.IOException;

public class EscrituraArchivo {
public static void main(String[] args) {
try {
FileWriter writer = new FileWriter("archivo.txt");
writer.write("Hola, mundo!\n");
```

```
writer.write("Esto es una prueba.");
writer.close();
System.out.println("Archivo escrito correctamente.");
} catch (IOException e) {
System.out.println("Error al escribir el archivo.");
e.printStackTrace();
}
}
}
```

LEER UN ARCHIVO DE TEXTO:

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class LecturaArchivo {
public static void main(String[] args) {
try {
FileReader reader = new FileReader("archivo.txt");
BufferedReader br = new BufferedReader(reader);
String linea;
while ((linea = br.readLine()) != null) {
System.out.println(linea);
}
br.close();
} catch (IOException e) {
System.out.println("Error al leer el archivo.");
e.printStackTrace();
}
}
}
```

7.3 CLASES FILE, FILEREADER, FILEWRITER

La clase File representa una ruta de archivo o directorio. No permite leer o escribir directamente, pero es útil para verificar existencia, tamaño, ruta, etc.

USO BÁSICO DE LA CLASE FILE:

```
import java.io.File;

public class UsoFile {
public static void main(String[] args) {
File archivo = new File("archivo.txt");

    if (archivo.exists()) {
        System.out.println("Nombre: " + archivo.getName());
        System.out.println("Ruta absoluta: " + archivo.getAbsolutePath());
        System.out.println("Escribible: " + archivo.canWrite());
        System.out.println("Legible: " + archivo.canRead());
        System.out.println("Tamaño: " + archivo.length() + " bytes");
```

```
    } else {
        System.out.println("El archivo no existe.");
    }
}
```

}

FILEREADER y FILEWRITER:

- FileReader: clase para leer archivos de texto carácter a carácter.
- FileWriter: clase para escribir caracteres en archivos de texto.

Ejemplo combinado:

import java.io.*;

public class ArchivoTexto {
public static void main(String[] args) {
try {
FileWriter fw = new FileWriter("demo.txt");
fw.write("Primera línea.\nSegunda línea.");
fw.close();

```
        FileReader fr = new FileReader("demo.txt");
        int c;
        while ((c = fr.read()) != -1) {
            System.out.print((char)c);
        }
        fr.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

}

7.4 SERIALIZACIÓN DE OBJETOS

La serialización es el proceso de convertir un objeto en una secuencia de bytes para poder guardarlo en un archivo o transmitirlo. Java facilita esto usando las clases ObjectOutputStream y ObjectInputStream.

REQUISITOS:

- La clase debe implementar la interfaz Serializable.

EJEMPLO DE SERIALIZACIÓN Y DESERIALIZACIÓN:

import java.io.*;

class Persona implements Serializable {
private static final long serialVersionUID = 1L;
String nombre;
int edad;

```
public Persona(String nombre, int edad) {
    this.nombre = nombre;
    this.edad = edad;
```

}

}

public class SerializarObjeto {
public static void main(String[] args) {
Persona p1 = new Persona("Ana", 30);

```
    // Serializar objeto
    try {
        FileOutputStream fileOut = new FileOutputStream("persona.ser");
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(p1);
        out.close();
        fileOut.close();
        System.out.println("Objeto serializado correctamente.");
    } catch (IOException i) {
        i.printStackTrace();
    }

    // Deserializar objeto
    try {
        FileInputStream fileIn = new FileInputStream("persona.ser");
        ObjectInputStream in = new ObjectInputStream(fileIn);
        Persona p2 = (Persona) in.readObject();
        in.close();
        fileIn.close();

        System.out.println("Nombre: " + p2.nombre);
        System.out.println("Edad: " + p2.edad);
    } catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
    }
}
```

}

NOTAS:

- serialVersionUID se usa para asegurar compatibilidad entre versiones de la clase.
- Todos los atributos también deben ser serializables o marcarse como transient.

CONCLUSIÓN

En este capítulo hemos aprendido a usar la clase Scanner para entrada estándar, a manipular archivos mediante File, FileReader y FileWriter, y a serializar objetos con ObjectOutputStream. Estas herramientas son esenciales para manejar datos persistentes y comunicar programas con el usuario o con otros sistemas.

Capítulo 9: Clases utilitarias del API de Java

Java proporciona un conjunto de clases utilitarias en su API que permiten realizar operaciones comunes de forma sencilla y eficiente. Estas clases abarcan operaciones matemáticas, manipulación de fechas y tiempos, colecciones, y tareas programadas, entre otras. Este capítulo explora las clases más representativas de estas utilidades.

9.1 Clases utilitarias: Math, Arrays, Objects, Collections

- Clase Math

La clase Math proporciona métodos estáticos para operaciones matemáticas como potencias, raíces, redondeo, funciones trigonométricas y logarítmicas. Algunos métodos útiles:

Math.abs(x) devuelve el valor absoluto
Math.pow(a, b) devuelve a elevado a la potencia b
Math.sqrt(x) devuelve la raíz cuadrada
Math.max(a, b) y Math.min(a, b) comparan dos valores
Math.round(x) redondea al entero más cercano
Math.random() devuelve un número aleatorio entre 0.0 y 1.0

Ejemplo:
double raiz = Math.sqrt(25); // 5.0
int maximo = Math.max(10, 20); // 20
double aleatorio = Math.random(); // valor entre 0.0 y 1.0

- Clase Arrays

La clase Arrays contiene métodos estáticos para trabajar con arreglos (arrays). Permite ordenarlos, buscarlos, copiarlos y compararlos.

Arrays.sort(array) ordena un array
Arrays.binarySearch(array, clave) busca un valor en un array ordenado
Arrays.equals(arr1, arr2) compara si dos arrays son iguales
Arrays.fill(array, valor) llena un array con un valor
Arrays.copyOf(array, longitud) copia un array

Ejemplo:
int[] numeros = {5, 3, 8};
Arrays.sort(numeros); // ahora: {3, 5, 8}
int pos = Arrays.binarySearch(numeros, 5); // devuelve 1

- Clase Objects

La clase Objects proporciona métodos para trabajar con objetos y realizar comprobaciones de nulidad y comparaciones seguras.

Objects.equals(a, b) compara dos objetos incluso si alguno es null
Objects.requireNonNull(obj) lanza NullPointerException si el objeto es null
Objects.hash(...) calcula el hashCode de varios objetos
Objects.toString(obj, "valor por defecto") convierte un objeto a String

Ejemplo:
String a = null;
String b = "hola";
boolean sonIguales = Objects.equals(a, b); // false

- Clase Collections

La clase Collections contiene métodos estáticos para manipular colecciones como listas, conjuntos y mapas.

Collections.sort(lista) ordena una lista
Collections.reverse(lista) invierte una lista
Collections.shuffle(lista) mezcla aleatoriamente una lista
Collections.max(lista), Collections.min(lista) obtienen el máximo y mínimo
Collections.frequency(lista, elemento) cuenta cuántas veces aparece un valor
Collections.unmodifiableList(lista) crea una lista inmutable

Ejemplo:
List lista = Arrays.asList(1, 2, 3, 4);
Collections.shuffle(lista); // cambia el orden aleatoriamente

9.2 Date, LocalDate, LocalTime, LocalDateTime

- Clase Date (java.util.Date)

Es una clase antigua que representa una fecha y hora. Hoy día se considera obsoleta en muchos casos y ha sido reemplazada por las clases del paquete java.time.

Date fecha = new Date();
System.out.println(fecha); // fecha y hora actual

Se pueden usar métodos como getTime() o setTime() para manejar fechas en milisegundos, pero es preferible usar LocalDateTime para nuevas aplicaciones.

- Clase LocalDate (java.time.LocalDate)

Representa una fecha sin hora. Útil para fechas de calendario como cumpleaños o eventos.

LocalDate hoy = LocalDate.now();
LocalDate nacimiento = LocalDate.of(1990, 5, 12);
LocalDate mañana = hoy.plusDays(1);
LocalDate añoPasado = hoy.minusYears(1);

Métodos comunes:
getYear(), getMonth(), getDayOfMonth(), isLeapYear(), isBefore(), isAfter()

- Clase LocalTime (java.time.LocalTime)

Representa solo una hora sin fecha.

LocalTime ahora = LocalTime.now();
LocalTime horaFija = LocalTime.of(14, 30);
LocalTime enUnaHora = ahora.plusHours(1);

- Clase LocalDateTime (java.time.LocalDateTime)

Combina una fecha y una hora.

```
LocalDateTime fechaHora = LocalDateTime.now();
LocalDateTime evento = LocalDateTime.of(2025, 12, 25, 20, 0);
LocalDateTime futuro = fechaHora.plusDays(5).minusHours(2);
```

Métodos útiles:
getYear(), getMonth(), getDayOfWeek(), plusMinutes(), minusDays()

9.3 Formato de fechas y horas

Para mostrar o interpretar fechas con un formato personalizado se usa la clase DateTimeFormatter del paquete java.time.format.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate fecha = LocalDate.of(2025, 7, 5);
String texto = fecha.format(formatter); // "05/07/2025"
```

También se puede analizar un texto para convertirlo en fecha:

```
String texto = "15-08-2024";
DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd-MM-yyyy");
LocalDate fechaConvertida = LocalDate.parse(texto, formato);
```

Formatos comunes:
"yyyy-MM-dd" (ISO estándar)
"dd/MM/yyyy"
"HH:mm:ss"
"dd MMMM yyyy" (por ejemplo "05 julio 2025")

Para fechas y horas completas:

```
LocalDateTime ahora = LocalDateTime.now();
DateTimeFormatter f = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");
String salida = ahora.format(f); // por ejemplo "05-07-2025 14:30"
```

9.4 Temporizadores y tareas programadas

Java permite ejecutar tareas con retardo o repetición usando clases como Timer, TimerTask y ScheduledExecutorService.

- Timer y TimerTask (java.util)

```
Timer timer = new Timer();
TimerTask tarea = new TimerTask() {
public void run() {
System.out.println("Ejecutando tarea...");
}
};
timer.schedule(tarea, 3000); // se ejecuta una vez después de 3 segundos
```

También puede ejecutarse de forma periódica:

```
timer.scheduleAtFixedRate(tarea, 0, 5000); // cada 5 segundos
```

- ScheduledExecutorService (java.util.concurrent)

Más moderno y robusto que Timer.

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
Runnable tarea = () -> System.out.println("Tarea ejecutada");
scheduler.schedule(tarea, 2, TimeUnit.SECONDS); // se ejecuta después de 2 segundos
```

También se puede repetir periódicamente:

```
scheduler.scheduleAtFixedRate(tarea, 0, 10, TimeUnit.SECONDS);
```

Este método es preferido en aplicaciones concurrentes y multiusuario, ya que maneja mejor los errores y excepciones.

Resumen del capítulo:

- La clase Math permite realizar operaciones matemáticas de forma simple.
- Las clases Arrays y Collections simplifican el trabajo con estructuras de datos.
- Las clases Date, LocalDate, LocalTime y LocalDateTime permiten trabajar con fechas y tiempos de forma moderna y flexible.
- DateTimeFormatter permite personalizar el formato de entrada y salida de fechas.
- Las tareas programadas se pueden gestionar con Timer o con ScheduledExecutorService, que es más recomendable.

Capítulo 10 - Programación concurrente y multihilo en Java

10.1 Conceptos de concurrencia

La programación concurrente es un paradigma que permite la ejecución de múltiples tareas o procesos de manera simultánea, mejorando la eficiencia y el rendimiento de los programas, especialmente en sistemas con múltiples núcleos. La concurrencia se puede lograr a través de la multitarea (multitasking), que puede ser a nivel de proceso o a nivel de hilo (thread). En Java, los hilos permiten ejecutar múltiples tareas dentro del mismo proceso de forma paralela o pseudo-paralela.

Un hilo (thread) es la unidad más pequeña de procesamiento que puede ser ejecutada de manera concurrente con otros hilos. Todos los programas Java comienzan con un hilo principal (main thread) que puede crear y gestionar hilos adicionales para realizar trabajos paralelos.

Los beneficios de la programación concurrente incluyen:

- Mejor aprovechamiento del hardware moderno con múltiples núcleos.
- Mayor capacidad de respuesta en aplicaciones interactivas.
- Posibilidad de realizar múltiples tareas en segundo plano sin bloquear el flujo principal del programa.

Sin embargo, la programación concurrente también implica desafíos como la sincronización de datos compartidos y la detección de errores como condiciones de carrera y bloqueos mutuos.

10.2 Clase Thread y Runnable

Java proporciona dos formas principales para crear hilos: extendiendo la clase Thread o implementando la interfaz Runnable.

1. Extendiendo la clase Thread:

Cuando se extiende la clase Thread, se sobrescribe el método run() que contiene el código que se ejecutará en paralelo.

Ejemplo:

```
class MiHilo extends Thread {
public void run() {
for (int i = 0; i < 5; i++) {
System.out.println("Hilo: " + i);
}
}
}
```

```
public class Principal {
public static void main(String[] args) {
MiHilo h = new MiHilo();
h.start(); // Inicia el hilo
}
}
```

2. Implementando la interfaz Runnable:

Esta es una opción más flexible porque permite que la clase herede de otras clases.

Ejemplo:

```
class Tarea implements Runnable {
public void run() {
for (int i = 0; i < 5; i++) {
System.out.println("Runnable: " + i);
}
}
}
```

```
public class Principal {
public static void main(String[] args) {
Thread hilo = new Thread(new Tarea());
hilo.start(); // Inicia el hilo
}
}
```

La diferencia clave entre `run()` y `start()` es que `start()` crea un nuevo hilo de ejecución mientras que `run()` ejecuta el código en el mismo hilo actual.

10.3 Sincronización

Cuando varios hilos acceden a recursos compartidos (como variables, archivos o estructuras de datos), es necesario garantizar que estos accesos se realicen de forma ordenada para evitar resultados inconsistentes. Esto se logra mediante la sincronización.

Java proporciona la palabra clave synchronized para evitar que múltiples hilos accedan a una sección crítica del código simultáneamente.

Ejemplo de método sincronizado:

```
class Contador {
private int cuenta = 0;

public synchronized void incrementar() {
    cuenta++;
}

public int obtenerCuenta() {
    return cuenta;
}

}
```

Otra opción es usar bloques sincronizados:

```
synchronized(objeto) {
// sección crítica
}
```

También existen herramientas avanzadas en el paquete java.util.concurrent como ReentrantLock, que permite mayor control sobre la sincronización, incluyendo bloqueo condicional y desbloqueo manual.

10.4 Ejecutores y Callable

El paquete java.util.concurrent proporciona una forma más flexible y moderna de gestionar hilos mediante el uso de ejecutores (Executors). La interfaz ExecutorService permite gestionar un grupo de hilos (thread pool), reutilizar hilos y evitar la creación manual de nuevos hilos.

Ejemplo básico con ExecutorService:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Tarea implements Runnable {
public void run() {
System.out.println("Tarea ejecutada por: " + Thread.currentThread().getName());
}
}

public class Principal {
public static void main(String[] args) {
ExecutorService executor = Executors.newFixedThreadPool(3);
for (int i = 0; i < 10; i++) {
executor.submit(new Tarea());
}
executor.shutdown();
}
}
```

Callable es similar a Runnable pero puede retornar un resultado y lanzar excepciones. Se usa junto con Future para obtener el resultado de una tarea asincrónica.

Ejemplo con Callable:

```
import java.util.concurrent.*;

class MiCallable implements Callable {
public Integer call() throws Exception {
return 123;
}
}

public class Principal {
public static void main(String[] args) throws Exception {
ExecutorService executor = Executors.newSingleThreadExecutor();
Future futuro = executor.submit(new MiCallable());
System.out.println("Resultado: " + futuro.get());
executor.shutdown();
}
}
```

10.5 Problemas comunes: race conditions, deadlocks

1. Condiciones de carrera (Race Conditions):

Ocurren cuando dos o más hilos acceden a una variable compartida sin sincronización adecuada, y al menos uno de los accesos es una escritura. Esto puede producir resultados impredecibles y errores difíciles de depurar.

Ejemplo típico:

```
class Contador {
private int cuenta = 0;

public void incrementar() {
    cuenta = cuenta + 1; // No es atómica
}

public int obtenerCuenta() {
    return cuenta;
}

}
```

Si múltiples hilos llaman a incrementar() sin sincronización, el valor final puede no ser correcto.

Solución: usar métodos sincronizados o variables atómicas como AtomicInteger.

2. Deadlocks (bloqueos mutuos):

Ocurren cuando dos o más hilos esperan mutuamente por un recurso que el otro posee, provocando un ciclo de espera infinita.

Ejemplo:

```
class Recurso {
public synchronized void usar(Recurso otro) {
System.out.println("Usando recurso y esperando otro");
otro.completar();
}

public synchronized void completar() {
    System.out.println("Completado");
}

}

public class Principal {
public static void main(String[] args) {
Recurso r1 = new Recurso();
Recurso r2 = new Recurso();

    Thread t1 = new Thread(() -> r1.usar(r2));
    Thread t2 = new Thread(() -> r2.usar(r1));

    t1.start();
    t2.start();
}
```

}

Ambos hilos quedan esperando indefinidamente porque cada uno tiene un recurso y necesita el otro.

Soluciones comunes para evitar deadlocks:

- Adquirir los recursos siempre en el mismo orden.
- Usar timeouts.
- Emplear estructuras de control de concurrencia avanzadas (Locks, semáforos, etc.).

Capítulo 11: Desarrollo de interfaces gráficas con Swing

11.1 Introducción a Swing

Swing es un conjunto de bibliotecas gráficas de Java que permite crear interfaces gráficas de usuario (GUI) para aplicaciones de escritorio. Swing forma parte del paquete javax.swing y es una mejora del toolkit AWT (Abstract Window Toolkit), ofreciendo mayor flexibilidad y una apariencia más moderna. Swing está completamente escrito en Java, lo que permite que las interfaces sean multiplataforma.

Swing proporciona una gran variedad de componentes como botones, etiquetas, cuadros de texto, listas, menús, tablas y mucho más. Además, se puede personalizar fácilmente el aspecto y comportamiento de los componentes.

Ventajas de Swing:

- Independencia de plataforma.
- Amplio conjunto de componentes.
- Soporte para temas (Look and Feel).
- Capacidad de personalización.
- Uso de contenedores y layout managers para organizar los elementos visuales.

11.2 Ventanas, paneles, botones, etiquetas

La clase base para crear una ventana es JFrame. Es el contenedor principal en una aplicación Swing. Otros contenedores comunes incluyen JPanel para agrupar componentes y facilitar su organización.

Ejemplo básico con JFrame, JButton y JLabel:

import javax.swing.*;

public class VentanaBasica {
public static void main(String[] args) {
JFrame ventana = new JFrame("Mi primera ventana");
ventana.setSize(300, 200);
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```
    JPanel panel = new JPanel();

    JLabel etiqueta = new JLabel("Hola, mundo!");
    JButton boton = new JButton("Pulsar");

    panel.add(etiqueta);
    panel.add(boton);

    ventana.add(panel);
    ventana.setVisible(true);
}
```

}

Componentes principales:

- JFrame: ventana principal.

- JPanel: panel para contener otros componentes.
- JLabel: muestra texto no editable.
- JButton: botón que puede generar acciones.

11.3 Campos de texto, áreas de texto, menús

Los campos de texto permiten al usuario introducir texto en una sola línea (JTextField) o en varias líneas (JTextArea). Los menús permiten organizar acciones y comandos dentro de la interfaz.

Ejemplo con JTextField, JTextArea y JMenu:

```
import javax.swing.;
import java.awt.;

public class InterfazTextoMenu {
public static void main(String[] args) {
JFrame frame = new JFrame("Texto y Menús");
frame.setSize(400, 300);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JTextField campoTexto = new JTextField(20);
    JTextArea areaTexto = new JTextArea(5, 20);

    JMenuBar barraMenu = new JMenuBar();
    JMenu menuArchivo = new JMenu("Archivo");
    JMenuItem itemSalir = new JMenuItem("Salir");

    menuArchivo.add(itemSalir);
    barraMenu.add(menuArchivo);
    frame.setJMenuBar(barraMenu);

    JPanel panel = new JPanel();
    panel.add(new JLabel("Introduce texto:"));
    panel.add(campoTexto);
    panel.add(new JScrollPane(areaTexto));

    frame.add(panel);
    frame.setVisible(true);
}

}
```

Componentes adicionales:

- JTextField: entrada de texto en una sola línea.
- JTextArea: entrada de texto en varias líneas.
- JScrollPane: permite añadir barras de desplazamiento a componentes como JTextArea.
- JMenuBar, JMenu, JMenuItem: componentes para crear menús.

11.4 Layouts y organización de componentes

Swing proporciona varios "Layout Managers" para controlar la disposición de los componentes dentro de un contenedor. Algunos de los más comunes son:

- FlowLayout: disposición en fila.

- BorderLayout: divide el contenedor en cinco áreas (NORTH, SOUTH, EAST, WEST, CENTER).
- GridLayout: cuadrícula de celdas iguales.
- BoxLayout: disposición en una sola línea vertical u horizontal.

Ejemplo con diferentes layouts:

import javax.swing.*;
import java.awt.*;

public class EjemploLayouts {
public static void main(String[] args) {
JFrame frame = new JFrame("Layouts");
frame.setSize(400, 300);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```
    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());

    panel.add(new JButton("Norte"), BorderLayout.NORTH);
    panel.add(new JButton("Sur"), BorderLayout.SOUTH);
    panel.add(new JButton("Este"), BorderLayout.EAST);
    panel.add(new JButton("Oeste"), BorderLayout.WEST);
    panel.add(new JButton("Centro"), BorderLayout.CENTER);

    frame.add(panel);
    frame.setVisible(true);
}
```

}

Usar el layout correcto ayuda a crear interfaces más organizadas y adaptables.

11.5 Eventos y manejo de acciones

La interacción del usuario con los componentes se maneja mediante eventos. Por ejemplo, hacer clic en un botón genera un evento que puede ser capturado y procesado mediante un listener.

Para los botones, se utiliza ActionListener:

import javax.swing.*;
import java.awt.event.*;

public class EventoBoton {
public static void main(String[] args) {
JFrame frame = new JFrame("Eventos");
JButton boton = new JButton("Haz clic");

```
    boton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null, "¡Botón pulsado!");
        }
    });

    frame.add(boton);
    frame.setSize(200, 100);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    frame.setVisible(true);
}

}
```

Otros tipos de eventos comunes:

- MouseListener: eventos del ratón.
- KeyListener: eventos del teclado.
- WindowListener: eventos de la ventana (abrir, cerrar, minimizar).

Se pueden implementar usando clases anónimas, expresiones lambda (desde Java 8) o clases internas.

11.6 Tablas y listas

Swing ofrece componentes para mostrar y gestionar datos tabulares o listas de elementos. Las clases principales son JTable y JList.

Ejemplo de JList:

```
import javax.swing.*;

public class ListaEjemplo {
public static void main(String[] args) {
JFrame frame = new JFrame("Lista");
String[] elementos = {"Elemento 1", "Elemento 2", "Elemento 3"};
JList lista = new JList<>(elementos);
JScrollPane scroll = new JScrollPane(lista);

    frame.add(scroll);
    frame.setSize(200, 150);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}

}
```

Ejemplo de JTable:

```
import javax.swing.*;

public class TablaEjemplo {
public static void main(String[] args) {
JFrame frame = new JFrame("Tabla");

    String[] columnas = {"Nombre", "Edad", "Ciudad"};
    String[][] datos = {
        {"Ana", "25", "Madrid"},
        {"Luis", "30", "Barcelona"},
        {"Elena", "22", "Valencia"}
    };

    JTable tabla = new JTable(datos, columnas);
    JScrollPane scroll = new JScrollPane(tabla);

    frame.add(scroll);
```

```
        frame.setSize(300, 150);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

}
```

Resumen de componentes:

- JList: lista de elementos seleccionables.
- JTable: tabla con filas y columnas.
- JScrollPane: permite añadir scroll a listas y tablas.

Capítulo 12: Acceso a bases de datos con JDBC

12.1 Introducción a JDBC

JDBC (Java Database Connectivity) es una API de Java que permite a las aplicaciones Java conectarse y operar con bases de datos relacionales. JDBC proporciona un conjunto estándar de interfaces y clases para enviar instrucciones SQL, obtener resultados y manejar errores de bases de datos. Funciona como una capa intermedia entre una aplicación Java y una base de datos, haciendo posible la independencia del motor de base de datos al usar diferentes drivers.

Componentes clave de JDBC:

- DriverManager: Gestiona una lista de drivers de base de datos.
- Connection: Representa una conexión con una base de datos.
- Statement: Permite ejecutar consultas SQL simples.
- PreparedStatement: Ejecuta consultas SQL parametrizadas.
- ResultSet: Contiene los datos devueltos por una consulta.
- SQLException: Maneja los errores que se produzcan durante la operación con la base de datos.

12.2 Conexión con bases de datos

Para conectarse a una base de datos usando JDBC, se necesitan tres cosas: el driver JDBC correspondiente, la URL de la base de datos, y las credenciales de acceso.

Pasos para establecer la conexión:

1. Cargar el driver JDBC.
2. Crear la conexión mediante la clase DriverManager.
3. Gestionar excepciones con SQLException.

Ejemplo:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConexionBD {
public static void main(String[] args) {
String url = "jdbc:mysql://localhost:3306/empresa";
String usuario = "root";
String contrasena = "password";

    try {
        Connection conexion = DriverManager.getConnection(url, usuario,
contrasena);
        System.out.println("Conexión exitosa");
        conexion.close();
    } catch (SQLException e) {
        System.out.println("Error al conectar: " + e.getMessage());
    }
```

}

}

Nota: En versiones modernas de Java, no es necesario registrar el driver manualmente si está incluido en el classpath.

12.3 Sentencias SQL desde Java

Una vez establecida la conexión, se pueden ejecutar sentencias SQL usando las interfaces Statement o PreparedStatement.

El método executeQuery() se usa para sentencias SELECT, y el método executeUpdate() para INSERT, UPDATE y DELETE.

Ejemplo de SELECT:

```
import java.sql.*;

public class ConsultarEmpleados {
public static void main(String[] args) {
try {
Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/empresa",
"root", "password");
Statement stmt = conexion.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM empleados");
        while (rs.next()) {
            int id = rs.getInt("id");
            String nombre = rs.getString("nombre");
            System.out.println("ID: " + id + ", Nombre: " + nombre);
        }

        rs.close();
        stmt.close();
        conexion.close();
    } catch (SQLException e) {
        System.out.println("Error: " + e.getMessage());
    }
}

}
```

12.4 PreparedStatement y ResultSet

PreparedStatement permite ejecutar sentencias SQL con parámetros, aumentando la seguridad (prevención de SQL Injection) y el rendimiento.

Ejemplo de uso de PreparedStatement:

```
import java.sql.*;

public class BuscarEmpleado {
public static void main(String[] args) {
try {
Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/empresa",
"root", "password");
```

```
String sql = "SELECT * FROM empleados WHERE id = ?";
PreparedStatement pstmt = conexion.prepareStatement(sql);
pstmt.setInt(1, 1);
ResultSet rs = pstmt.executeQuery();
        if (rs.next()) {
            System.out.println("Nombre: " + rs.getString("nombre"));
        }

        rs.close();
        pstmt.close();
        conexion.close();
    } catch (SQLException e) {
        System.out.println("Error: " + e.getMessage());
    }
}

}
```

El objeto ResultSet proporciona métodos para acceder a los datos obtenidos como getInt(), getString(), getDouble(), etc.

12.5 Inserción, actualización y eliminación

Para realizar operaciones de modificación sobre la base de datos (INSERT, UPDATE, DELETE), se usa executeUpdate() que retorna el número de filas afectadas.

Ejemplo de inserción:

```
import java.sql.*;

public class InsertarEmpleado {
public static void main(String[] args) {
try {
Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/empresa",
"root", "password");
String sql = "INSERT INTO empleados (nombre, salario) VALUES (?, ?)";
PreparedStatement pstmt = conexion.prepareStatement(sql);
pstmt.setString(1, "Laura");
pstmt.setDouble(2, 35000);
int filas = pstmt.executeUpdate();
        System.out.println("Filas insertadas: " + filas);
        pstmt.close();
        conexion.close();
    } catch (SQLException e) {
        System.out.println("Error: " + e.getMessage());
    }
}

}
```

Ejemplo de actualización:

```
String sql = "UPDATE empleados SET salario = ? WHERE id = ?";
PreparedStatement pstmt = conexion.prepareStatement(sql);
pstmt.setDouble(1, 40000);
```

pstmt.setInt(2, 1);
pstmt.executeUpdate();

Ejemplo de eliminación:

String sql = "DELETE FROM empleados WHERE id = ?";
PreparedStatement pstmt = conexion.prepareStatement(sql);
pstmt.setInt(1, 2);
pstmt.executeUpdate();

12.6 Manejo de transacciones

Por defecto, JDBC realiza auto-commit después de cada sentencia. Para realizar transacciones manuales se puede desactivar el auto-commit.

Ejemplo de transacción:

import java.sql.*;

public class TransaccionEjemplo {
public static void main(String[] args) {
try {
Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/empresa", "root", "password");
conexion.setAutoCommit(false);

```
        PreparedStatement pstmt1 = conexion.prepareStatement("UPDATE empleados 
SET salario = salario + 1000 WHERE id = ?");
        pstmt1.setInt(1, 1);
        pstmt1.executeUpdate();

        PreparedStatement pstmt2 = conexion.prepareStatement("UPDATE empleados 
SET salario = salario - 1000 WHERE id = ?");
        pstmt2.setInt(1, 2);
        pstmt2.executeUpdate();

        conexion.commit();
        System.out.println("Transacción completada con éxito");

        pstmt1.close();
        pstmt2.close();
        conexion.close();
    } catch (SQLException e) {
        System.out.println("Error en la transacción: " + e.getMessage());
        try {
            conexion.rollback();
            System.out.println("Transacción revertida");
        } catch (SQLException ex) {
            System.out.println("Error al hacer rollback: " + ex.getMessage());
        }
    }
}
```

}

12.7 Cierre y gestión de recursos

Es fundamental cerrar todos los recursos JDBC (ResultSet, Statement, Connection) una vez que han sido usados, para liberar recursos del sistema y evitar fugas de memoria.

Desde Java 7 se recomienda usar try-with-resources, que cierra automáticamente los objetos que implementan AutoCloseable.

Ejemplo con try-with-resources:

import java.sql.*;

public class ConsultaConTry {
public static void main(String[] args) {
String url = "jdbc:mysql://localhost:3306/empresa";
String user = "root";
String password = "password";

```
    String sql = "SELECT * FROM empleados";

    try (
        Connection conexion = DriverManager.getConnection(url, user, password);
        Statement stmt = conexion.createStatement();
        ResultSet rs = stmt.executeQuery(sql);
    ) {
        while (rs.next()) {
            System.out.println("Empleado: " + rs.getString("nombre"));
        }
    } catch (SQLException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
```

}

Resumen:

- JDBC es la API estándar de Java para trabajar con bases de datos relacionales.
- Se puede conectar a cualquier base de datos siempre que se tenga el driver adecuado.
- Statement y PreparedStatement permiten ejecutar sentencias SQL.
- ResultSet permite recorrer los resultados de las consultas.
- PreparedStatement mejora la seguridad y eficiencia.
- Las transacciones permiten ejecutar múltiples operaciones como una unidad lógica.
- El cierre adecuado de recursos es esencial para evitar problemas de rendimiento.

Este capítulo proporciona una base completa para interactuar con bases de datos usando JDBC. En siguientes capítulos se puede profundizar en conexión con pool de conexiones, ORMs como Hibernate o JPA, y acceso a bases de datos no relacionales.

Capítulo 13: Programación web con Java (introducción)

13.1 Java EE y Java Jakarta

Java EE (Java Platform, Enterprise Edition), actualmente conocido como Jakarta EE tras su transferencia a la Eclipse Foundation, es una plataforma robusta y estandarizada para el desarrollo de aplicaciones empresariales y web en Java. Jakarta EE ofrece un conjunto de especificaciones que extienden Java SE para facilitar la creación de aplicaciones distribuidas, transaccionales y escalables. Estas especificaciones incluyen tecnologías como Servlets, JSP (JavaServer Pages), JSF (JavaServer Faces), JPA (Java Persistence API), EJB (Enterprise JavaBeans) y muchas más.

Jakarta EE proporciona una arquitectura basada en componentes reutilizables y contenedores de servidor que gestionan servicios como la seguridad, el ciclo de vida de los objetos, la persistencia y la concurrencia. Uno de los beneficios principales es que el desarrollador puede enfocarse en la lógica de negocio sin preocuparse por los detalles de bajo nivel del servidor.

Los servidores de aplicaciones como GlassFish, WildFly o Payara implementan estas especificaciones y permiten desplegar aplicaciones web empresariales de forma sencilla.

13.2 Servlets y JSP

Los Servlets son componentes Java que se ejecutan en un servidor web o de aplicaciones. Actúan como controladores que procesan solicitudes HTTP y generan respuestas, típicamente en HTML. Un Servlet es una clase Java que extiende la clase HttpServlet e implementa métodos como doGet y doPost para manejar solicitudes GET y POST respectivamente.

JSP (JavaServer Pages) es una tecnología que permite insertar código Java directamente dentro de páginas HTML usando una sintaxis especial con etiquetas como <% ... %>. Esto facilita la generación dinámica de contenido HTML, ya que el código Java se traduce en un Servlet en tiempo de ejecución.

Los Servlets y JSP trabajan juntos en una arquitectura típica MVC (Modelo Vista Controlador), donde el Servlet actúa como el controlador que gestiona la lógica y la navegación, y las JSP como vistas que presentan la información al usuario.

Ejemplo básico de Servlet:

```
import java.io.;
import javax.servlet.;
import javax.servlet.http.*;

public class HolaMundoServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("");
out.println("Hola Mundo desde Servlet");
out.println("");
}
}
```

13.3 Configuración de un servidor Tomcat

Apache Tomcat es uno de los servidores de aplicaciones más populares para ejecutar Servlets y JSP. Para configurar Tomcat y empezar a trabajar con Java web, se siguen los siguientes pasos:

1. Descargar Tomcat desde el sitio oficial de Apache Tomcat.
2. Descomprimir el archivo en una carpeta de tu sistema.
3. Establecer la variable de entorno JAVA_HOME apuntando al JDK instalado.
4. Ejecutar el servidor con el script startup.bat (Windows) o startup.sh (Linux/Mac).
5. Acceder a la consola administrativa en http://localhost:8080

Para desplegar una aplicación web en Tomcat, se debe:

- Crear una estructura de proyecto que incluya un archivo web.xml dentro de la carpeta WEB-INF
- Incluir las clases compiladas en la carpeta WEB-INF/classes
- Empaquetar todo en un archivo WAR y copiarlo a la carpeta webapps de Tomcat

Ejemplo de estructura básica:

```
miapp/
├── index.jsp
└── WEB-INF/
├── web.xml
└── classes/
└── HolaMundoServlet.class
```

El archivo web.xml configura el mapeo del Servlet:

13.4 Formularios HTML y envío a Servlets

Una funcionalidad común en aplicaciones web es capturar datos del usuario mediante formularios HTML y enviarlos al servidor para ser procesados. Esto se hace mediante la etiqueta especificando el método (GET o POST) y la URL de destino que será manejada por un Servlet.

Ejemplo de formulario HTML:

Este formulario enviará el valor del campo "nombre" al Servlet mapeado en la URL "/procesar". El Servlet debe implementar el método doPost para capturar y procesar los datos.

Ejemplo de Servlet que recibe datos del formulario:

```
public class ProcesarFormulario extends HttpServlet {
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
String nombre = request.getParameter("nombre");
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("");
out.println("Bienvenido " + nombre + "");
```

out.println("");
}
}

13.5 Conexión de Servlets a bases de datos

Para que un Servlet interactúe con una base de datos, se utiliza JDBC (Java Database Connectivity). Esto permite establecer conexiones, ejecutar consultas SQL y procesar resultados desde Java.

Los pasos básicos son:

1. Cargar el driver JDBC adecuado
2. Establecer la conexión con la base de datos
3. Crear y ejecutar consultas
4. Procesar los resultados
5. Cerrar la conexión

Ejemplo de conexión de un Servlet a una base de datos MySQL:

import java.sql.*;*
import javax.servlet.;
import javax.servlet.http.*;*
import java.io.;

public class ConsultaUsuario extends HttpServlet {
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
String usuario = request.getParameter("usuario");
String clave = request.getParameter("clave");

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();

try {
  Class.forName("com.mysql.cj.jdbc.Driver");
  Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/miBD", "root",
"password");

  PreparedStatement stmt = con.prepareStatement("SELECT * FROM usuarios WHERE
nombre=? AND clave=?");
  stmt.setString(1, usuario);
  stmt.setString(2, clave);
  ResultSet rs = stmt.executeQuery();

  if (rs.next()) {
    out.println("<h2>Acceso concedido</h2>");
  } else {
    out.println("<h2>Acceso denegado</h2>");
  }

  con.close();
} catch (Exception e) {
  out.println("Error: " + e.getMessage());
}
```

```
}
}
```

Es importante cerrar siempre las conexiones para liberar recursos y evitar problemas de rendimiento.

Capítulo 14: Introducción a Spring Framework

Spring Framework es una de las plataformas más populares para el desarrollo de aplicaciones Java modernas, especialmente en el contexto de aplicaciones empresariales y web. Su diseño modular permite trabajar de manera eficiente con componentes como la inversión de control (IoC), la inyección de dependencias, la arquitectura MVC, y la persistencia de datos con JPA. En este capítulo exploraremos los conceptos esenciales de Spring, con un enfoque práctico y progresivo.

14.1 Inversión de control (IoC) y contenedores

Uno de los conceptos centrales de Spring es la Inversión de Control (IoC), también conocida como Inyección de Dependencias (DI). En lugar de que una clase cree sus propias dependencias, un contenedor de Spring las crea y las inyecta. Esto permite un código más flexible, desacoplado y fácil de testear.

El contenedor de Spring es el responsable de instanciar, configurar y ensamblar los objetos de la aplicación. Este contenedor se configura mediante archivos XML, anotaciones o clases Java. Los objetos gestionados por el contenedor se denominan beans.

Ejemplo básico de un bean:

```
public class ServicioSaludo {
public void saludar() {
System.out.println("Hola desde Spring!");
}
}
```

Configuración con anotación:

```
@Component
public class ServicioSaludo {
public void saludar() {
System.out.println("Hola desde Spring!");
}
}
```

Para que Spring detecte esta clase, se debe escanear el paquete mediante anotaciones como @ComponentScan.

14.2 Spring Boot y creación de proyectos

Spring Boot simplifica el desarrollo de aplicaciones Spring mediante configuración automática y un entorno de ejecución preconfigurado. Permite crear aplicaciones listas para producción con mínima configuración.

Para crear un proyecto Spring Boot se puede utilizar Spring Initializr (https://start.spring.io). Basta con seleccionar los siguientes parámetros:

- Lenguaje: Java
- Versión: 3.x (o la más reciente)
- Dependencias: Spring Web, Spring Data JPA, H2 Database (u otra base de datos), Spring Boot DevTools

Spring Boot proporciona un archivo principal con la clase con anotación @SpringBootApplication:

```
@SpringBootApplication
public class AplicacionPrincipal {
public static void main(String[] args) {
SpringApplication.run(AplicacionPrincipal.class, args);
}
}
```

Esta anotación engloba otras tres: @Configuration, @EnableAutoConfiguration y @ComponentScan.

14.3 Inyección de dependencias

Spring permite inyectar dependencias de varias formas: mediante constructor, métodos setter o directamente con anotaciones. Las anotaciones más comunes son @Autowired, @Component, @Service y @Repository.

Ejemplo con inyección por constructor:

```
@Component
public class ClienteServicio {
private final ServicioSaludo servicioSaludo;

@Autowired
public ClienteServicio(ServicioSaludo servicioSaludo) {
    this.servicioSaludo = servicioSaludo;
}

public void ejecutar() {
    servicioSaludo.saludar();
}

}
```

Spring detectará automáticamente el bean ServicioSaludo y lo inyectará en ClienteServicio.

Es posible usar perfiles (@Profile), ámbitos (@Scope) y cualificadores (@Qualifier) para controlar más finamente cómo se inyectan las dependencias.

14.4 Repositorios y controladores REST

Spring facilita la creación de APIs RESTful a través del módulo Spring Web. Los controladores REST se definen usando @RestController y @RequestMapping o @GetMapping, @PostMapping, etc.

Ejemplo de controlador REST:

```
@RestController
@RequestMapping("/saludos")
public class SaludoController {

@GetMapping
public String saludar() {
    return "Hola mundo desde Spring REST!";
}
```

```
}
```

Para manejar la capa de persistencia, se utilizan interfaces que extienden JpaRepository, las cuales son detectadas automáticamente gracias a la anotación @Repository.

Ejemplo:

```
@Entity
public class Usuario {
@Id
@GeneratedValue
private Long id;
private String nombre;
}
```

```
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
List findByNombre(String nombre);
}
```

Los métodos personalizados se construyen siguiendo una convención de nombres, y Spring implementa automáticamente las consultas.

14.5 JPA y conexión a base de datos

Spring Data JPA es un módulo que permite trabajar con bases de datos relacionales usando JPA (Java Persistence API). Spring se integra con proveedores como Hibernate para realizar el mapeo objeto-relacional (ORM).

La configuración de la base de datos puede realizarse en el archivo application.properties:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
```

Con esta configuración, Spring Boot crea una base de datos H2 en memoria. También se puede usar MySQL, PostgreSQL u otra base de datos.

La entidad se define con @Entity, y los campos se anotan con @Id, @GeneratedValue, @Column, etc.

Ejemplo:

```
@Entity
public class Producto {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String nombre;
private double precio;

}
```

Los repositorios se usan para acceder a los datos sin escribir código SQL:

```
public interface ProductoRepository extends JpaRepository<Producto, Long> {
List findByNombreContaining(String texto);
}
```

14.6 Arquitectura MVC en Spring

Spring MVC es un modelo de desarrollo basado en la arquitectura Modelo-Vista-Controlador. Esta arquitectura separa la lógica de negocio, la lógica de presentación y el control del flujo de la aplicación.

Modelo: Representa los datos y reglas de negocio (entidades, servicios).
Vista: Interfaz de usuario (Thymeleaf, JSP, HTML).
Controlador: Gestiona las peticiones y coordina la vista y el modelo.

Ejemplo de controlador MVC:

```
@Controller
public class ProductoControlador {

@Autowired
private ProductoRepository productoRepository;

@GetMapping("/productos")
public String listarProductos(Model model) {
    model.addAttribute("productos", productoRepository.findAll());
    return "productos"; // plantilla productos.html
}

}
```

El archivo HTML correspondiente puede utilizar el motor de plantillas Thymeleaf para mostrar los datos:

Spring MVC maneja automáticamente la conversión de datos, validaciones, gestión de sesiones y otros aspectos del desarrollo web moderno.

Conclusión

Spring Framework, junto con Spring Boot, ofrece un ecosistema completo para desarrollar aplicaciones Java modernas, modulares y mantenibles. Desde la inversión de control, pasando por la inyección de dependencias, hasta la creación de APIs REST y la persistencia con JPA, Spring permite construir aplicaciones robustas y escalables con poco código repetitivo. Comprender su arquitectura MVC facilita aún más el desarrollo de aplicaciones web limpias y bien estructuradas.

Capítulo 15: Pruebas y depuración en Java

15.1 Introducción a JUnit

JUnit es un framework para realizar pruebas unitarias en Java. Las pruebas unitarias consisten en verificar que unidades pequeñas de código, generalmente métodos, funcionan correctamente de manera aislada. JUnit proporciona anotaciones y métodos para definir y ejecutar pruebas de forma automatizada, facilitando la identificación de errores durante el desarrollo.

Para usar JUnit, se debe incluir la dependencia correspondiente en el proyecto. Por ejemplo, usando Maven:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.9.3</version>
  <scope>test</scope>
</dependency>
```

La versión más usada actualmente es JUnit 5, también conocida como Jupiter. Ofrece una API moderna, soporte para pruebas parametrizadas y mejor integración con IDEs y herramientas de construcción.

Una clase de prueba típica en JUnit 5 contiene métodos anotados con @Test que definen las pruebas. Por ejemplo:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CalculadoraTest {

    @Test
    void sumaDosNumeros() {
        Calculadora calc = new Calculadora();
        int resultado = calc.sumar(2, 3);
        assertEquals(5, resultado);
    }
}
```

15.2 Asserts y pruebas unitarias

Los asserts son afirmaciones que se usan para verificar que los resultados obtenidos coinciden con los esperados. JUnit ofrece varios métodos estáticos para hacer estas comprobaciones:

- assertEquals(expected, actual): Verifica que dos valores son iguales.
- assertTrue(condition): Verifica que una condición sea verdadera.
- assertFalse(condition): Verifica que una condición sea falsa.
- assertNull(object): Verifica que un objeto es null.
- assertNotNull(object): Verifica que un objeto no es null.
- assertThrows(exceptionClass, executable): Verifica que se lance una excepción esperada.

Por ejemplo:

```
@Test
void dividirPorCeroLanzaExcepcion() {
```

```
    Calculadora calc = new Calculadora();
    assertThrows(ArithmeticException.class, () -> calc.dividir(10, 0));
}
```

Las pruebas unitarias deben ser independientes, repetibles y rápidas. Es importante cubrir tanto casos normales como casos límite o excepcionales para asegurar la robustez del código.

15.3 Mocking con Mockito

Mockito es un framework para crear objetos simulados (mocks) en las pruebas unitarias. Los mocks permiten aislar la unidad bajo prueba simulando dependencias externas, como bases de datos, servicios web o componentes complejos.

Para usar Mockito, también se añade como dependencia:

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>5.4.0</version>
  <scope>test</scope>
</dependency>
```

Ejemplo básico de uso:

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;

class ServicioUsuarioTest {

    @Test
    void obtenerUsuarioPorId() {
        RepositorioUsuario repoMock = mock(RepositorioUsuario.class);
        Usuario usuarioSimulado = new Usuario(1, "Ana");
        when(repoMock.buscarPorId(1)).thenReturn(usuarioSimulado);

        ServicioUsuario servicio = new ServicioUsuario(repoMock);
        Usuario resultado = servicio.obtenerUsuario(1);

        assertEquals("Ana", resultado.getNombre());
        verify(repoMock).buscarPorId(1);
    }
}
```

Con Mockito se pueden definir comportamientos, verificar interacciones y lanzar excepciones simuladas para probar diferentes escenarios sin necesidad de acceso a recursos externos.

15.4 Técnicas de depuración en IDEs

La depuración es el proceso de encontrar y corregir errores en el código. Los IDEs modernos como IntelliJ IDEA, Eclipse o NetBeans ofrecen herramientas para depurar aplicaciones Java de forma visual.

Las técnicas principales incluyen:

- Puntos de interrupción (breakpoints): Pausan la ejecución del programa en una línea específica para inspeccionar variables y flujo.
- Inspección de variables: Permite ver el valor de variables en tiempo real durante la depuración.

- Paso a paso (step over, step into, step out): Controla la ejecución para avanzar línea a línea, entrar en métodos o salir de ellos.
- Evaluación de expresiones: Ejecutar expresiones arbitrarias para verificar resultados en tiempo de depuración.
- Visualización de la pila de llamadas (call stack): Muestra el orden de llamadas que llevaron al punto actual.

Por ejemplo, para depurar un método problemático:

1. Colocar un breakpoint en la línea donde sospechas el error.
2. Ejecutar la aplicación en modo depuración.
3. Cuando la ejecución se detenga, inspeccionar las variables.
4. Avanzar paso a paso para observar el comportamiento.
5. Modificar el código o corregir la lógica basado en lo encontrado.

15.5 Logs y seguimiento de errores

El registro de logs es fundamental para monitorear el comportamiento de la aplicación y diagnosticar problemas. Java dispone de varias bibliotecas para logging como java.util.logging, Log4j o SLF4J con Logback.

Un ejemplo simple con SLF4J y Logback:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class ServicioPago {

    private static final Logger logger =
LoggerFactory.getLogger(ServicioPago.class);

    public void procesarPago(double monto) {
        logger.info("Iniciando el proceso de pago por {}", monto);
        try {
            // lógica de pago
            logger.debug("Validando datos del pago");
            if (monto <= 0) {
                logger.error("Monto inválido: {}", monto);
                throw new IllegalArgumentException("Monto debe ser positivo");
            }
            // más lógica
            logger.info("Pago procesado exitosamente");
        } catch (Exception e) {
            logger.error("Error al procesar el pago", e);
        }
    }
}
```

Los niveles comunes de logs son:

- TRACE: Información muy detallada, para diagnóstico profundo.
- DEBUG: Información para depuración.
- INFO: Eventos importantes pero normales.

- WARN: Situaciones inesperadas pero no críticas.
- ERROR: Errores que afectan el funcionamiento.

Configurar adecuadamente el logging permite recolectar información útil para resolver errores, especialmente en entornos de producción donde no se puede usar la depuración interactiva.

Capítulo 16: Buenas prácticas y patrones de diseño en Java

16.1 Principios SOLID

Los principios SOLID son un conjunto de cinco directrices para el diseño de software orientado a objetos que facilitan la creación de sistemas mantenibles, escalables y robustos. Cada letra de SOLID representa un principio específico:

- **S: Principio de Responsabilidad Única (Single Responsibility Principle - SRP):** Una clase debe tener una sola razón para cambiar, es decir, debe encargarse de una única responsabilidad o función. Esto mejora la cohesión y facilita el mantenimiento.
- **O: Principio de Abierto/Cerrado (Open/Closed Principle - OCP):** Las entidades de software (clases, módulos, funciones) deben estar abiertas para su extensión pero cerradas para su modificación. Se busca extender el comportamiento sin alterar el código existente, por ejemplo, usando herencia o composición.
- **L: Principio de Sustitución de Liskov (Liskov Substitution Principle - LSP):** Los objetos de una clase derivada deben poder sustituir a objetos de la clase base sin alterar el correcto funcionamiento del programa. Esto garantiza que las subclases mantengan el contrato establecido por la superclase.
- **I: Principio de Segregación de Interfaces (Interface Segregation Principle - ISP):** Es mejor tener varias interfaces específicas y pequeñas que una interfaz general y grande. Los clientes no deben verse obligados a depender de interfaces que no utilizan.
- **D: Principio de Inversión de Dependencias (Dependency Inversion Principle - DIP):** Los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones. Las abstracciones no deben depender de detalles, sino que los detalles deben depender de abstracciones. Esto promueve el desacoplamiento.

Implementar estos principios ayuda a producir código más limpio, entendible y flexible, que puede adaptarse fácilmente a cambios futuros.

16.2 Refactorización de código

La refactorización es el proceso de mejorar la estructura interna del código sin modificar su comportamiento externo. Es una práctica fundamental para mantener la calidad del software y evitar la degradación del código (conocida como "deuda técnica").

Algunos objetivos de la refactorización incluyen:

- Mejorar la legibilidad y comprensión del código.
- Reducir la duplicación.
- Simplificar estructuras complejas.
- Mejorar la modularidad y reutilización.
- Facilitar la detección y corrección de errores.

Ejemplos comunes de refactorización en Java:

- **Extraer método:** Dividir métodos largos en métodos más pequeños con nombres descriptivos.
- **Renombrar variables y métodos:** Usar nombres claros y significativos.
- **Eliminar código muerto:** Quitar código que no se utiliza.
- **Reemplazar condicionales complejos:** Usar polimorfismo o patrones de diseño para simplificar decisiones.
- **Introducir objetos para representar conceptos:** Crear nuevas clases para encapsular datos y comportamientos relacionados.

Herramientas como IntelliJ IDEA o Eclipse ofrecen soporte para refactorización automática, lo que facilita la aplicación segura de estos cambios.

16.3 Documentación con Javadoc

Javadoc es la herramienta estándar de Java para generar documentación en formato HTML a partir de comentarios especiales dentro del código fuente. Una buena documentación facilita que otros desarrolladores comprendan y utilicen el código.

Comentarios Javadoc se colocan antes de clases, métodos, atributos y paquetes, y siguen un formato específico:

```
/**
 * Descripción general de la clase o método.
 *
 * @param nombreParametro Descripción del parámetro.
 * @return Descripción del valor retornado.
 * @throws Excepcion Descripción de la excepción lanzada.
 */
```

Ejemplo:

```
/**
 * Calcula la suma de dos enteros.
 *
 * @param a El primer número entero.
 * @param b El segundo número entero.
 * @return La suma de a y b.
 */
public int sumar(int a, int b) {
    return a + b;
}
```

Javadoc permite documentar:

- **@param:** Parámetros de métodos.
- **@return:** Valor devuelto por un método.
- **@throws o @exception:** Excepciones que puede lanzar un método.
- **@see:** Referencias a otras clases o métodos.
- **@deprecated:** Indicar que un método o clase está obsoleto.
- **@author:** Autor del código.

- **@version:** Versión del código.

Mantener la documentación actualizada es clave para un proyecto sostenible.

16.4 Patrones de diseño: Singleton, Factory, Observer, MVC

Los patrones de diseño son soluciones probadas a problemas comunes en el diseño de software. Facilitan la comunicación entre desarrolladores y promueven buenas prácticas.

- **Singleton:**

El patrón Singleton asegura que una clase tenga una única instancia y proporciona un punto global de acceso a ella. Es útil para recursos compartidos como configuraciones o conexiones a base de datos.

Ejemplo básico en Java:

```
public class Singleton {
    private static Singleton instancia;

    private Singleton() {
        // Constructor privado evita instanciación externa
    }

    public static synchronized Singleton getInstance() {
        if (instancia == null) {
            instancia = new Singleton();
        }
        return instancia;
    }
}
```

- **Factory (Fábrica):**

El patrón Factory define una interfaz para crear objetos, pero permite a las subclases decidir qué clase instanciar. Se usa para encapsular la creación de objetos, evitando dependencia directa con clases concretas.

Ejemplo:

```
public interface Animal {
    void hacerSonido();
}

public class Perro implements Animal {
    public void hacerSonido() {
        System.out.println("Guau");
    }
}

public class Gato implements Animal {
    public void hacerSonido() {
        System.out.println("Miau");
    }
}

public class AnimalFactory {
    public static Animal crearAnimal(String tipo) {
        if (tipo.equalsIgnoreCase("perro")) {
            return new Perro();
        } else if (tipo.equalsIgnoreCase("gato")) {
```

```
            return new Gato();
        }
        return null;
    }
}
```

- **Observer (Observador):**

El patrón Observer define una dependencia uno a muchos entre objetos, de modo que cuando uno cambia su estado, todos sus dependientes son notificados y actualizados automáticamente. Se usa en sistemas de eventos o interfaces gráficas.

Ejemplo básico:

```
import java.util.ArrayList;
import java.util.List;

interface Observador {
    void actualizar(String mensaje);
}

class Sujeto {
    private List<Observador> observadores = new ArrayList<>();

    public void agregarObservador(Observador o) {
        observadores.add(o);
    }

    public void eliminarObservador(Observador o) {
        observadores.remove(o);
    }

    public void notificarObservadores(String mensaje) {
        for (Observador o : observadores) {
            o.actualizar(mensaje);
        }
    }
}

class ObservadorConcreto implements Observador {
    private String nombre;

    public ObservadorConcreto(String nombre) {
        this.nombre = nombre;
    }

    public void actualizar(String mensaje) {
        System.out.println(nombre + " recibió mensaje: " + mensaje);
    }
}
```

- **MVC (Modelo-Vista-Controlador):**

MVC es un patrón arquitectónico que separa la aplicación en tres componentes:

- **Modelo:** Gestiona los datos y la lógica de negocio.
- **Vista:** Presenta los datos al usuario.
- **Controlador:** Recibe las acciones del usuario y coordina el modelo y la vista.

Esta separación mejora la modularidad y facilita el mantenimiento y pruebas.

Ejemplo simplificado:

```java
// Modelo
public class Modelo {
    private String dato;

    public String getDato() {
        return dato;
    }

    public void setDato(String dato) {
        this.dato = dato;
    }
}

// Vista
public class Vista {
    public void mostrarDato(String dato) {
        System.out.println("Dato: " + dato);
    }
}

// Controlador
public class Controlador {
    private Modelo modelo;
    private Vista vista;

    public Controlador(Modelo modelo, Vista vista) {
        this.modelo = modelo;
        this.vista = vista;
    }

    public void setDato(String dato) {
        modelo.setDato(dato);
    }

    public void actualizarVista() {
        vista.mostrarDato(modelo.getDato());
    }
}
```

Para concluir, adoptar buenas prácticas y patrones de diseño en Java es fundamental para crear software limpio, eficiente y fácil de mantener. El conocimiento y aplicación de principios como SOLID, junto con la refactorización continua y documentación clara, sumado al uso adecuado de patrones como Singleton, Factory, Observer y MVC, aporta calidad y profesionalismo a los proyectos.

Capítulo 17: Herramientas y entornos de desarrollo en Java

17.1 Uso de IDEs: IntelliJ IDEA, Eclipse, NetBeans

En el desarrollo moderno de aplicaciones Java, los entornos de desarrollo integrados (IDEs) son fundamentales para mejorar la productividad, la organización y la depuración del código. Los tres IDEs más populares para Java son IntelliJ IDEA, Eclipse y NetBeans.

IntelliJ IDEA es un IDE desarrollado por JetBrains, conocido por su potente asistencia de código, refactorización avanzada, integración con sistemas de control de versiones y herramientas de construcción, así como soporte para múltiples frameworks y tecnologías. Su versión Community es gratuita y suficiente para muchos proyectos, mientras que la versión Ultimate ofrece funcionalidades avanzadas para desarrollo empresarial.

Eclipse es uno de los IDEs más antiguos y populares para Java. Es open source y cuenta con un ecosistema extenso de plugins que permiten ampliar sus funcionalidades para diferentes lenguajes y frameworks. Eclipse es muy usado en proyectos grandes y en ambientes corporativos por su flexibilidad.

NetBeans es un IDE también open source, patrocinado inicialmente por Sun Microsystems y luego por Apache. Tiene integración nativa con Maven y Gradle, y facilita la creación rápida de proyectos Java con plantillas y asistentes. Es muy amigable para principiantes y ofrece soporte para desarrollo web, móvil y de escritorio.

En general, estos IDEs permiten editar código con resaltado sintáctico, autocompletado, navegación rápida, integración con sistemas de control de versiones, ejecución y depuración de aplicaciones, gestión de dependencias y pruebas unitarias.

17.2 Uso de Git y GitHub

Git es un sistema de control de versiones distribuido que permite gestionar el historial de cambios en el código fuente. Es fundamental para trabajar en equipo, ya que permite que múltiples desarrolladores colaboren sin perder versiones anteriores ni sobrescribir el trabajo de otros.

GitHub es una plataforma web que aloja repositorios Git y facilita la colaboración mediante funcionalidades como issues, pull requests, revisiones de código, wikis y gestión de proyectos.

En un proyecto Java, se recomienda inicializar un repositorio Git desde el inicio. Se pueden realizar commits frecuentes con mensajes descriptivos para documentar los cambios. Para subir el código a GitHub, se crea un repositorio remoto y se conecta al local mediante comandos git remote add. Luego, se usa git push para subir los commits.

Git permite crear ramas (branches) para trabajar en nuevas funcionalidades o correcciones sin afectar la rama principal. Al terminar, se pueden fusionar con merge o rebase. Es importante resolver conflictos de código cuando dos ramas modifican las mismas líneas.

Los IDEs como IntelliJ IDEA, Eclipse y NetBeans ofrecen integración con Git y GitHub, lo que facilita operaciones como commit, push, pull y merge desde la interfaz gráfica sin usar la consola.

17.3 Automatización con Maven y Gradle

Maven y Gradle son herramientas de automatización de construcción que gestionan la compilación, ejecución de pruebas, generación de documentación, empaquetado y manejo de dependencias en proyectos Java.

Maven utiliza un archivo de configuración llamado pom.xml donde se declaran las dependencias, plugins y configuraciones del proyecto. Su estructura basada en convenciones facilita la estandarización y portabilidad entre proyectos. Al ejecutar comandos como mvn clean install, Maven descarga dependencias del repositorio central, compila el código, ejecuta pruebas y genera artefactos empaquetados.

Gradle es una herramienta más moderna y flexible que usa scripts en Groovy o Kotlin DSL. Permite configuraciones dinámicas y mejores tiempos de construcción gracias a su sistema de cacheo y construcción incremental. Su archivo build.gradle declara las dependencias y tareas. Gradle es especialmente popular en proyectos Android, pero también en aplicaciones Java de servidor y escritorio.

Ambas herramientas permiten definir tareas personalizadas, gestionar proyectos multi-módulo, y ejecutar fases específicas como pruebas unitarias, análisis estático, generación de informes y despliegue.

17.4 Construcción de proyectos modulares

Con Java 9 se introdujo el sistema de módulos, que permite dividir grandes aplicaciones en módulos independientes y encapsulados. Esto mejora la organización, reutilización y seguridad del código.

Un módulo Java se define con un archivo module-info.java donde se declara el nombre del módulo, los paquetes que exporta y las dependencias con otros módulos. Por ejemplo:

```
module com.ejemplo.app {
requires java.sql;
exports com.ejemplo.app.servicios;
}
```

La construcción modular facilita la carga dinámica de módulos, evita conflictos de versiones y reduce el tamaño final de las aplicaciones.

Maven y Gradle soportan proyectos modulares. En Maven, se pueden configurar múltiples módulos en un proyecto padre con un archivo pom.xml principal y varios pom.xml en subdirectorios. Gradle permite definir subproyectos y sus dependencias mediante configuraciones en settings.gradle y build.gradle.

17.5 Integración continua

La integración continua (CI) es una práctica de desarrollo donde los cambios en el código se integran de forma frecuente y automática, permitiendo detectar errores rápidamente y asegurar la calidad.

Para proyectos Java, se configuran pipelines de CI que ejecutan automáticamente compilaciones, pruebas unitarias, análisis estático de código, empaquetado y despliegue a entornos de prueba cada vez que se hace un commit o push al repositorio.

Herramientas populares de CI incluyen Jenkins, GitHub Actions, GitLab CI/CD, Travis CI y CircleCI. Estas herramientas se configuran con scripts o archivos YAML que definen las etapas, comandos y condiciones del pipeline.

Un pipeline básico de CI para un proyecto Java con Maven podría incluir:

- Checkout del código desde GitHub
- Instalación de Java y Maven en el agente
- Ejecución de mvn clean test para compilar y ejecutar pruebas
- Análisis de resultados y notificación a desarrolladores

La CI mejora la colaboración, reduce defectos y acelera el ciclo de desarrollo.

Resumen

Este capítulo presentó las principales herramientas y entornos para el desarrollo en Java, incluyendo IDEs potentes como IntelliJ IDEA, Eclipse y NetBeans, el control de versiones con Git y GitHub, la automatización con Maven y Gradle, la construcción modular con Java 9 y posteriores, y las prácticas de integración continua para asegurar calidad y eficiencia. Estas herramientas y prácticas son esenciales para el desarrollo profesional y colaborativo de proyectos Java modernos.

Capítulo 18: Proyecto Final en Java

18.1 Diseño y planificación del proyecto

El primer paso para la realización de un proyecto en Java es su diseño y planificación. Esta fase es fundamental para definir el alcance, los objetivos, las funcionalidades y las restricciones del sistema. Para ello se recomienda:

- Definir el problema a resolver y los requisitos funcionales y no funcionales.
- Analizar el entorno donde se desplegará la aplicación.
- Realizar un diseño arquitectónico inicial que incluya diagramas UML como diagramas de clases, de casos de uso y de secuencia.
- Planificar las tareas, asignar tiempos estimados y definir las fases de desarrollo.
- Seleccionar las herramientas, frameworks y librerías que se usarán.

Un buen diseño ayuda a evitar problemas durante el desarrollo y facilita la integración de módulos y futuras ampliaciones.

18.2 Desarrollo en fases (modelo incremental)

El desarrollo incremental consiste en construir el proyecto por etapas o fases, donde en cada fase se añade funcionalidad hasta completar el sistema. Este modelo permite:

- Entregar versiones funcionales parciales para pruebas tempranas.
- Detectar errores o desviaciones de requisitos antes de completar el proyecto.
- Adaptar el diseño según el feedback recibido.

Para aplicar este modelo en Java:

- Dividir el proyecto en módulos o funcionalidades independientes.
- Implementar primero las funciones básicas y críticas.
- Probar y validar cada incremento antes de continuar.
- Añadir mejoras y funcionalidades secundarias en incrementos sucesivos.

Por ejemplo, en un sistema de gestión, el primer incremento puede incluir el registro de usuarios y autenticación, el segundo la gestión de datos básicos y el tercero funcionalidades avanzadas como reportes o gráficos.

18.3 Pruebas y validación del sistema

Las pruebas son esenciales para garantizar que el sistema funciona correctamente, cumple con los requisitos y es estable. En Java existen varias estrategias:

- Pruebas unitarias: prueban métodos o clases individuales. Se recomienda usar frameworks como JUnit para automatizarlas.
- Pruebas de integración: verifican que los módulos funcionan bien en conjunto.
- Pruebas funcionales: comprueban que el sistema cumple con los requisitos desde la perspectiva del usuario.

- Pruebas de aceptación: realizadas por usuarios finales o clientes para validar el producto.

Se deben definir casos de prueba con entradas esperadas y resultados esperados. Además es importante realizar pruebas de rendimiento y manejo de errores. La validación asegura que el software cumple las expectativas y es confiable.

18.4 Documentación técnica y de usuario

La documentación es una parte clave del proyecto, pues facilita el mantenimiento, la ampliación y el uso del sistema. Se recomienda generar dos tipos principales:

- Documentación técnica: incluye diagramas de diseño, descripción de la arquitectura, explicación del código fuente, configuración del entorno y manuales para desarrolladores.
- Documentación de usuario: describe cómo instalar, configurar y usar el sistema. Debe ser clara y contener ejemplos y capturas de pantalla.

Es conveniente mantener la documentación actualizada durante todo el ciclo de vida del proyecto y usar herramientas que permitan su generación automática cuando sea posible.

18.5 Presentación del proyecto

La presentación final del proyecto debe resumir todo el trabajo realizado y mostrar los resultados. Debe incluir:

- Objetivos y alcance del proyecto.
- Diseño y arquitectura general.
- Funcionalidades implementadas.
- Demostración práctica o prototipo funcionando.
- Resultados de las pruebas y validación.
- Conclusiones y posibles mejoras futuras.

Se recomienda preparar material visual como diapositivas, diagramas y demos en vivo. La presentación debe ser clara, ordenada y enfocada a la audiencia, que puede ser un profesor, cliente o equipo de trabajo.

Capítulo 19: Recursos adicionales

19.1 Libros recomendados

Para profundizar en Java y mejorar tus habilidades, existen muchos libros que cubren desde los fundamentos hasta temas avanzados. Algunos de los más recomendados son:

- "Effective Java" de Joshua Bloch: considerado un libro imprescindible para desarrolladores Java, enseña buenas prácticas, patrones y trucos para escribir código eficiente y mantenible.
- "Java: The Complete Reference" de Herbert Schildt: cubre todo el lenguaje Java, incluyendo las librerías estándar y conceptos clave, ideal para principiantes y como referencia.
- "Head First Java" de Kathy Sierra y Bert Bates: un libro con un enfoque visual y didáctico que facilita la comprensión de conceptos básicos y orientados a objetos.
- "Java Concurrency in Practice" de Brian Goetz: para entender en profundidad la programación concurrente y multihilo en Java.
- "Spring in Action" de Craig Walls: para aprender sobre el popular framework Spring, muy usado en desarrollo backend con Java.
- "Java Performance: The Definitive Guide" de Scott Oaks: orientado a la optimización y el análisis del rendimiento de aplicaciones Java.

19.2 Documentación oficial de Java

La documentación oficial es la fuente más confiable y actualizada para aprender y consultar sobre Java. Está disponible en la página oficial de Oracle:

- Documentación del JDK: https://docs.oracle.com/en/java/javase/

Aquí encontrarás:

- Tutoriales para principiantes y avanzados.
- API completa de las clases y paquetes estándar.
- Guías sobre herramientas del JDK como javac, javadoc, jvm, etc.
- Documentación específica para cada versión de Java.

Además, el JDK incluye un conjunto de ejemplos y herramientas para probar el código directamente.

19.3 Foros y comunidades

Participar en comunidades es fundamental para resolver dudas, compartir conocimientos y mantenerse actualizado. Algunas de las comunidades y foros más importantes para desarrolladores Java son:

- Stack Overflow (https://stackoverflow.com): la comunidad de preguntas y respuestas más grande, con muchas consultas resueltas sobre Java.
- Reddit - r/java (https://www.reddit.com/r/java/): un foro activo para discusión sobre noticias, frameworks y problemas de Java.

- Oracle Java Community (https://community.oracle.com/tech/developers/categories/java): foro oficial para desarrolladores Java.
- JavaRanch (https://coderanch.com): comunidad de desarrolladores Java donde se pueden hacer preguntas y participar en discusiones.
- GitHub (https://github.com): no es un foro, pero muchos proyectos Java de código abierto se alojan aquí, es útil para estudiar código real y contribuir.

19.4 Certificaciones Java

Las certificaciones son una forma formal de validar tus conocimientos y habilidades en Java. Algunas certificaciones importantes son:

- Oracle Certified Associate, Java SE Programmer (OCAJP): certificación de nivel inicial que cubre conceptos básicos del lenguaje.
- Oracle Certified Professional, Java SE Programmer (OCPJP): certificación avanzada para programadores Java con experiencia.
- Oracle Certified Expert en Java EE Web Component Developer: para desarrolladores con conocimientos en tecnologías web y Java EE.
- Oracle Certified Master, Java SE Developer (OCMJD): certificación de nivel experto para profesionales con amplio dominio de Java.

Las certificaciones pueden mejorar tu perfil profesional y abrir puertas en el mercado laboral.

19.5 Rutas de especialización: backend, Android, data science, enterprise

Java es un lenguaje versátil que permite especializarse en diferentes áreas de desarrollo:

- Backend: desarrollo de aplicaciones servidor con frameworks como Spring, Hibernate, y microservicios. Aquí se crean APIs, sistemas de negocio y bases de datos.
- Android: Java es uno de los lenguajes principales para programar aplicaciones móviles Android, aunque ahora Kotlin está ganando terreno. La especialización incluye desarrollo UI, manejo de sensores y optimización móvil.
- Data Science: aunque no es el lenguaje más usado en ciencia de datos, Java tiene librerías para procesamiento de datos, machine learning y Big Data como Weka, Deeplearning4j o Apache Hadoop.
- Enterprise: desarrollo de aplicaciones empresariales complejas usando Java EE, incluyendo EJB, JPA, JMS, y servicios web. Se enfocan en sistemas escalables, seguridad y transacciones distribuidas.

Cada ruta requiere aprender herramientas, frameworks y conceptos específicos. Explorar estas especializaciones ayuda a orientar tu carrera según tus intereses y el mercado.

Ejercicios

Ejercicio 1
Enunciado: Escribe un programa que imprima "Hola, mundo".
Solución:

```
public class Ejercicio1 {
public static void main(String[] args) {
System.out.println("Hola, mundo");
}
}
```

Explicación: Se usa `System.out.println` para mostrar texto en consola.

Ejercicio 2
Enunciado: Escribe un programa que sume dos números enteros.
Solución:

```
public class Ejercicio2 {
public static void main(String[] args) {
int a = 5;
int b = 3;
int suma = a + b;
System.out.println("La suma es: " + suma);
}
}
```

Explicación: Declaramos dos variables enteras y las sumamos.

Ejercicio 3
Enunciado: Escribe un programa que calcule el área de un círculo.
Solución:

```
public class Ejercicio3 {
public static void main(String[] args) {
double radio = 4.5;
double area = Math.PI * radio * radio;
System.out.println("Área del círculo: " + area);
}
}
```

Explicación: Se usa la fórmula A = π * r^2.

Ejercicio 4
Enunciado: Escribe un programa que intercambie los valores de dos variables.
Solución:
```
public class Ejercicio4 {
public static void main(String[] args) {
int x = 10;
int y = 20;
int temp = x;
x = y;
y = temp;
System.out.println("x: " + x + ", y: " + y);
}
}
```
Explicación: Se usa una variable temporal para intercambiar los valores.

Ejercicio 5
Enunciado: Escribe un programa que determine si un número es par o impar.
Solución:
```
public class Ejercicio5 {
public static void main(String[] args) {
int numero = 7;
if (numero % 2 == 0) {
System.out.println("Es par");
} else {
System.out.println("Es impar");
}
}
}
```
Explicación: Usamos el operador módulo (%) para verificar si hay residuo.

Ejercicio 6
Enunciado: Escribe un programa que muestre los números del 1 al 10.
Solución:
```
public class Ejercicio6 {
public static void main(String[] args) {
for (int i = 1; i <= 10; i++) {
System.out.println(i);
}
}
}
```
Explicación: Usamos un bucle for para contar del 1 al 10.

Ejercicio 7
Enunciado: Escribe un programa que calcule el factorial de un número.
Solución:

```
public class Ejercicio7 {
public static void main(String[] args) {
int numero = 5;
int factorial = 1;
for (int i = 1; i <= numero; i++) {
factorial *= i;
}
System.out.println("Factorial: " + factorial);
}
}
```

Explicación: El factorial se obtiene multiplicando números consecutivos.

Ejercicio 8
Enunciado: Escribe un programa que convierta grados Celsius a Fahrenheit.
Solución:

```
public class Ejercicio8 {
public static void main(String[] args) {
double celsius = 25;
double fahrenheit = (celsius * 9/5) + 32;
System.out.println("Fahrenheit: " + fahrenheit);
}
}
```

Explicación: Se usa la fórmula F = C * 9/5 + 32.

Ejercicio 9
Enunciado: Escribe un programa que imprima los primeros 5 múltiplos de 3.
Solución:

```
public class Ejercicio9 {
public static void main(String[] args) {
for (int i = 1; i <= 5; i++) {
System.out.println(i * 3);
}
}
}
```

Explicación: Se multiplica 3 por cada número del 1 al 5.

Ejercicio 10
Enunciado: Escribe un programa que calcule el promedio de tres números.
Solución:
```
public class Ejercicio10 {
public static void main(String[] args) {
double a = 8.5, b = 7.0, c = 9.5;
double promedio = (a + b + c) / 3;
System.out.println("Promedio: " + promedio);
}
}
```
Explicación: Sumamos los tres valores y dividimos entre 3.

Ejercicio 11
Enunciado: Escribe un programa que determine si un número es positivo, negativo o cero.
Solución:

```
public class Ejercicio11 {
public static void main(String[] args) {
int numero = -4;
if (numero > 0) {
System.out.println("Positivo");
} else if (numero < 0) {
System.out.println("Negativo");
} else {
System.out.println("Cero");
}
}
}
```

Explicación: Se usan condicionales para comparar el número con 0.

Ejercicio 12
Enunciado: Escribe un programa que imprima los números pares entre 1 y 20.
Solución:

```
public class Ejercicio12 {
public static void main(String[] args) {
for (int i = 2; i <= 20; i += 2) {
System.out.println(i);
}
}
}
```

Explicación: Incrementamos el contador de 2 en 2 para imprimir solo pares.

Ejercicio 13
Enunciado: Escribe un programa que pida un número y muestre su tabla de multiplicar del 1 al 10.
Solución:

```
import java.util.Scanner;
public class Ejercicio13 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
for (int i = 1; i <= 10; i++) {
System.out.println(n + " x " + i + " = " + (n * i));
}
sc.close();
}
}
```

Explicación: Leemos un número con Scanner y usamos un bucle para multiplicar.

Ejercicio 14
Enunciado: Escribe un programa que cuente cuántos números del 1 al 100 son múltiplos de 5.
Solución:

```
public class Ejercicio14 {
public static void main(String[] args) {
int contador = 0;
for (int i = 1; i <= 100; i++) {
if (i % 5 == 0) {
contador++;
}
}
System.out.println("Múltiplos de 5: " + contador);
}
}
```

Explicación: Contamos con un if dentro del bucle usando módulo.

Ejercicio 15
Enunciado: Escribe un programa que imprima la suma de los números del 1 al 50.
Solución:

```
public class Ejercicio15 {
public static void main(String[] args) {
int suma = 0;
for (int i = 1; i <= 50; i++) {
suma += i;
}
System.out.println("Suma: " + suma);
}
}
```

Explicación: Sumamos todos los números en el bucle.

Ejercicio 16
Enunciado: Escribe un programa que reciba un número y diga si es primo o no.
Solución:

```
import java.util.Scanner;
public class Ejercicio16 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
boolean esPrimo = true;
if (n <= 1) esPrimo = false;
for (int i = 2; i <= Math.sqrt(n); i++) {
if (n % i == 0) {
esPrimo = false;
break;
}
}
if (esPrimo) System.out.println("Es primo");
else System.out.println("No es primo");
sc.close();
}
}
```

Explicación: Probamos divisores desde 2 hasta raíz cuadrada.

Ejercicio 17
Enunciado: Escribe un programa que imprima los primeros 10 números Fibonacci.
Solución:
```
public class Ejercicio17 {
public static void main(String[] args) {
int a = 0, b = 1;
System.out.println(a);
System.out.println(b);
for (int i = 3; i <= 10; i++) {
int c = a + b;
System.out.println(c);
a = b;
b = c;
}
}
}
```
Explicación: La serie suma los dos números anteriores.

Ejercicio 18
Enunciado: Escribe un programa que convierta una cantidad dada en dólares a euros, considerando 1 USD = 0.85 EUR.
Solución:
```
import java.util.Scanner;
public class Ejercicio18 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Dólares: ");
double usd = sc.nextDouble();
double eur = usd * 0.85;
System.out.println("Euros: " + eur);
sc.close();
}
}
```
Explicación: Multiplicamos por el tipo de cambio dado.

Ejercicio 19
Enunciado: Escribe un programa que determine si un año es bisiesto.
Solución:
```
import java.util.Scanner;
public class Ejercicio19 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Año: ");
int año = sc.nextInt();
boolean esBisiesto = (año % 4 == 0 && año % 100 != 0) || (año % 400 == 0);
if (esBisiesto) System.out.println("Es bisiesto");
else System.out.println("No es bisiesto");
sc.close();
}
}
```
Explicación: La regla para años bisiestos es esa.

Ejercicio 20
Enunciado: Escribe un programa que imprima los números del 10 al 1 en orden descendente.
Solución:
```
public class Ejercicio20 {
public static void main(String[] args) {
for (int i = 10; i >= 1; i--) {
System.out.println(i);
}
}
}
```
Explicación: El bucle for cuenta hacia atrás decrementando.

Ejercicio 21
Enunciado: Escribe un programa que reciba un nombre y lo salude.
Solución:

```
import java.util.Scanner;
public class Ejercicio21 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Ingrese su nombre: ");
String nombre = sc.nextLine();
System.out.println("Hola, " + nombre + "!");
sc.close();
}
}
```

Explicación: Se usa Scanner para leer texto y concatenar en el saludo.

Ejercicio 22
Enunciado: Escribe un programa que imprima los números del 1 al 100, pero para múltiplos de 3 imprima "Fizz" y para múltiplos de 5 "Buzz".
Solución:

```
public class Ejercicio22 {
public static void main(String[] args) {
for (int i = 1; i <= 100; i++) {
if (i % 3 == 0) System.out.println("Fizz");
else if (i % 5 == 0) System.out.println("Buzz");
else System.out.println(i);
}
}
}
```

Explicación: Usamos condicionales para verificar los múltiplos.

Ejercicio 23
Enunciado: Escribe un programa que convierta una cadena a mayúsculas.
Solución:

```
public class Ejercicio23 {
public static void main(String[] args) {
String texto = "hola mundo";
System.out.println(texto.toUpperCase());
}
}
```

Explicación: `toUpperCase()` convierte texto a mayúsculas.

Ejercicio 24
Enunciado: Escribe un programa que sume los dígitos de un número entero.
Solución:
```
public class Ejercicio24 {
public static void main(String[] args) {
int numero = 1234;
int suma = 0;
while (numero > 0) {
suma += numero % 10;
numero /= 10;
}
System.out.println("Suma de dígitos: " + suma);
}
}
```
Explicación: Se usa módulo y división para extraer y sumar dígitos.

Ejercicio 25
Enunciado: Escribe un programa que muestre si un carácter es vocal o consonante.
Solución:
```
public class Ejercicio25 {
public static void main(String[] args) {
char c = 'a';
if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' ||
c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U') {
System.out.println("Vocal");
} else {
System.out.println("Consonante");
}
}
}
```
Explicación: Comparamos el carácter con las vocales.

Ejercicio 26
Enunciado: Escribe un programa que calcule la potencia de un número base elevado a un exponente entero positivo.
Solución:

```
public class Ejercicio26 {
public static void main(String[] args) {
int base = 2;
int exponente = 5;
int resultado = 1;
for (int i = 1; i <= exponente; i++) {
resultado *= base;
}
System.out.println("Resultado: " + resultado);
}
}
```

Explicación: Multiplicamos la base por sí misma exponente veces.

Ejercicio 27
Enunciado: Escribe un programa que imprima los primeros 15 números impares.
Solución:

```
public class Ejercicio27 {
public static void main(String[] args) {
int count = 0;
int num = 1;
while (count < 15) {
System.out.println(num);
num += 2;
count++;
}
}
}
```

Explicación: Los números impares incrementan de 2 en 2 desde 1.

Ejercicio 28
Enunciado: Escribe un programa que reciba una frase y cuente cuántas palabras tiene.
Solución:

```
import java.util.Scanner;
public class Ejercicio28 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Ingrese una frase: ");
String frase = sc.nextLine();
String[] palabras = frase.trim().split("\s+");
System.out.println("Número de palabras: " + palabras.length);
sc.close();
}
}
```

Explicación: Se usa split para dividir la frase en palabras.

Ejercicio 29
Enunciado: Escribe un programa que imprima un triángulo rectángulo de asteriscos de altura 5.
Solución:

```
public class Ejercicio29 {
public static void main(String[] args) {
for (int i = 1; i <= 5; i++) {
for (int j = 1; j <= i; j++) {
System.out.print("*");
}
System.out.println();
}
}
}
```

Explicación: Usamos dos bucles anidados para imprimir el patrón.

Ejercicio 30
Enunciado: Escribe un programa que invierta una cadena dada.
Solución:

```
public class Ejercicio30 {
public static void main(String[] args) {
String texto = "Java";
String invertido = "";
for (int i = texto.length() - 1; i >= 0; i--) {
invertido += texto.charAt(i);
}
System.out.println("Invertido: " + invertido);
}
}
```

Explicación: Recorremos la cadena desde el final hacia el principio.

Ejercicio 31
Enunciado: Escribe un programa que calcule la suma de los números pares entre 1 y 100.
Solución:

```
public class Ejercicio31 {
public static void main(String[] args) {
int suma = 0;
for (int i = 2; i <= 100; i += 2) {
suma += i;
}
System.out.println("Suma pares: " + suma);
}
}
```

Explicación: Se suma solo números pares incrementando de 2 en 2.

Ejercicio 32
Enunciado: Escribe un programa que determine si una cadena es un palíndromo (se lee igual al derecho y al revés).
Solución:

```
public class Ejercicio32 {
public static void main(String[] args) {
String texto = "anilina";
String invertido = "";
for (int i = texto.length() - 1; i >= 0; i--) {
invertido += texto.charAt(i);
}
if (texto.equalsIgnoreCase(invertido)) {
System.out.println("Es palíndromo");
} else {
System.out.println("No es palíndromo");
}
}
}
```

Explicación: Comparamos la cadena con su versión invertida ignorando mayúsculas.

Ejercicio 33
Enunciado: Escribe un programa que imprima la suma de los números impares del 1 al 50.
Solución:

```
public class Ejercicio33 {
public static void main(String[] args) {
int suma = 0;
for (int i = 1; i <= 50; i += 2) {
suma += i;
}
System.out.println("Suma impares: " + suma);
}
}
```

Explicación: Se suma solo números impares incrementando de 2 en 2.

Ejercicio 34
Enunciado: Escribe un programa que calcule el máximo común divisor (MCD) de dos números.
Solución:

```
public class Ejercicio34 {
public static void main(String[] args) {
int a = 24, b = 36;
while (b != 0) {
int temp = b;
b = a % b;
a = temp;
}
System.out.println("MCD: " + a);
}
}
```

Explicación: Se usa el algoritmo de Euclides para calcular el MCD.

Ejercicio 35
Enunciado: Escribe un programa que genere un número aleatorio entre 1 y 100.
Solución:

```
public class Ejercicio35 {
public static void main(String[] args) {
int aleatorio = (int)(Math.random() * 100) + 1;
System.out.println("Número aleatorio: " + aleatorio);
}
}
```

Explicación: `Math.random()` genera un número decimal entre 0 y 1.

Ejercicio 36
Enunciado: Escribe un programa que muestre la fecha y hora actual.
Solución:

```
import java.time.LocalDateTime;
public class Ejercicio36 {
public static void main(String[] args) {
LocalDateTime ahora = LocalDateTime.now();
System.out.println("Fecha y hora: " + ahora);
}
}
```

Explicación: Se usa la clase `LocalDateTime` para obtener fecha y hora actual.

Ejercicio 37
Enunciado: Escribe un programa que convierta un número entero a binario.
Solución:

```
public class Ejercicio37 {
public static void main(String[] args) {
int numero = 13;
String binario = Integer.toBinaryString(numero);
System.out.println("Binario: " + binario);
}
}
```

Explicación: `Integer.toBinaryString` convierte el número a cadena binaria.

Ejercicio 38
Enunciado: Escribe un programa que reciba una frase y cuente cuántas vocales tiene.
Solución:

```
public class Ejercicio38 {
public static void main(String[] args) {
String frase = "Hola mundo";
int contador = 0;
for (int i = 0; i < frase.length(); i++) {
char c = Character.toLowerCase(frase.charAt(i));
if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
contador++;
}
}
System.out.println("Número de vocales: " + contador);
}
}
```

Explicación: Se verifica cada carácter y se cuenta si es vocal.

Ejercicio 39
Enunciado: Escribe un programa que muestre los primeros 20 números de la serie 2, 4, 8, 16,... (potencias de 2).
Solución:

```
public class Ejercicio39 {
public static void main(String[] args) {
int valor = 2;
for (int i = 1; i <= 20; i++) {
System.out.println(valor);
valor *= 2;
}
}
}
```

Explicación: Se multiplica por 2 en cada iteración para generar la serie.

Ejercicio 40
Enunciado: Escribe un programa que reciba un número y muestre si es múltiplo de 7.
Solución:

```
import java.util.Scanner;
public class Ejercicio40 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
if (n % 7 == 0) {
System.out.println("Es múltiplo de 7");
} else {
System.out.println("No es múltiplo de 7");
}
sc.close();
}
}
```

Explicación: Usamos módulo para determinar si el residuo es cero.

Ejercicio 41
Enunciado: Escribe un programa que imprima los números del 1 al 50 que sean múltiplos de 4.
Solución:
```
public class Ejercicio41 {
public static void main(String[] args) {
for (int i = 1; i <= 50; i++) {
if (i % 4 == 0) {
System.out.println(i);
}
}
}
}
```
Explicación: Se verifica el residuo de la división por 4 para filtrar los múltiplos.

Ejercicio 42
Enunciado: Escribe un programa que reciba dos números y muestre cuál es el mayor.
Solución:
```
import java.util.Scanner;
public class Ejercicio42 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Ingrese el primer número: ");
int a = sc.nextInt();
System.out.print("Ingrese el segundo número: ");
int b = sc.nextInt();
if (a > b) {
System.out.println(a + " es mayor");
} else if (b > a) {
System.out.println(b + " es mayor");
} else {
System.out.println("Ambos son iguales");
}
sc.close();
}
}
```
Explicación: Se comparan los dos números usando condicionales.

Ejercicio 43
Enunciado: Escribe un programa que calcule el área de un círculo dado su radio.
Solución:

```
import java.util.Scanner;
public class Ejercicio43 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Radio: ");
double radio = sc.nextDouble();
double area = Math.PI * radio * radio;
System.out.println("Área: " + area);
sc.close();
}
}
```

Explicación: Se aplica la fórmula área = $\pi * r^2$ usando Math.PI.

Ejercicio 44
Enunciado: Escribe un programa que reciba una cadena y muestre su longitud.
Solución:

```
import java.util.Scanner;
public class Ejercicio44 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Ingrese una cadena: ");
String texto = sc.nextLine();
System.out.println("Longitud: " + texto.length());
sc.close();
}
}
```

Explicación: Se usa el método length() para obtener el número de caracteres.

Ejercicio 45
Enunciado: Escribe un programa que imprima los números del 10 al 1 en orden descendente.
Solución:

```
public class Ejercicio45 {
public static void main(String[] args) {
for (int i = 10; i >= 1; i--) {
System.out.println(i);
}
}
}
```

Explicación: Se usa un bucle for que decrece desde 10 hasta 1.

Ejercicio 46
Enunciado: Escribe un programa que determine si un número es positivo, negativo o cero.
Solución:

```
import java.util.Scanner;
public class Ejercicio46 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
if (n > 0) {
System.out.println("Positivo");
} else if (n < 0) {
System.out.println("Negativo");
} else {
System.out.println("Cero");
}
sc.close();
}
}
```

Explicación: Se evalúa el signo del número con condicionales.

Ejercicio 47
Enunciado: Escribe un programa que reciba un número y calcule su factorial.
Solución:

```
import java.util.Scanner;
public class Ejercicio47 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
int factorial = 1;
for (int i = 1; i <= n; i++) {
factorial *= i;
}
System.out.println("Factorial: " + factorial);
sc.close();
}
}
```

Explicación: Se multiplica desde 1 hasta n para calcular el factorial.

Ejercicio 48
Enunciado: Escribe un programa que convierta una temperatura de grados Celsius a Fahrenheit.
Solución:

```
import java.util.Scanner;
public class Ejercicio48 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Grados Celsius: ");
double celsius = sc.nextDouble();
double fahrenheit = (celsius * 9 / 5) + 32;
System.out.println("Grados Fahrenheit: " + fahrenheit);
sc.close();
}
}
```

Explicación: Se usa la fórmula para convertir Celsius a Fahrenheit.

Ejercicio 49
Enunciado: Escribe un programa que imprima los números pares entre dos números dados por el usuario.
Solución:

```
import java.util.Scanner;
public class Ejercicio49 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número inicial: ");
int inicio = sc.nextInt();
System.out.print("Número final: ");
int fin = sc.nextInt();
for (int i = inicio; i <= fin; i++) {
if (i % 2 == 0) {
System.out.println(i);
}
}
sc.close();
}
}
```

Explicación: Se recorren los números entre inicio y fin y se imprimen los pares.

Ejercicio 50
Enunciado: Escribe un programa que reciba un número y muestre si es primo o no.
Solución:

```
import java.util.Scanner;
public class Ejercicio50 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
boolean esPrimo = true;
if (n <= 1) {
esPrimo = false;
} else {
for (int i = 2; i <= Math.sqrt(n); i++) {
if (n % i == 0) {
esPrimo = false;
break;
}
}
}
if (esPrimo) {
System.out.println("Es primo");
} else {
System.out.println("No es primo");
}
sc.close();
}
}
```

Explicación: Se verifica si el número tiene divisores distintos de 1 y él mismo.

Ejercicio 51
Enunciado: Escribe un programa que imprima los números del 1 al 100, pero para múltiplos de 3 imprima "Fizz", para múltiplos de 5 "Buzz" y para múltiplos de ambos "FizzBuzz".
Solución:

```
public class Ejercicio51 {
public static void main(String[] args) {
for (int i = 1; i <= 100; i++) {
if (i % 3 == 0 && i % 5 == 0) {
System.out.println("FizzBuzz");
} else if (i % 3 == 0) {
System.out.println("Fizz");
} else if (i % 5 == 0) {
System.out.println("Buzz");
} else {
System.out.println(i);
}
}
}
}
```

Explicación: Se usan condicionales para verificar múltiplos de 3 y 5.

Ejercicio 52
Enunciado: Escribe un programa que calcule la suma de los dígitos de un número entero.
Solución:

```
import java.util.Scanner;
public class Ejercicio52 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
int suma = 0;
while (n != 0) {
suma += n % 10;
n /= 10;
}
System.out.println("Suma de dígitos: " + suma);
sc.close();
}
}
```

Explicación: Se extraen y suman los dígitos usando módulo y división.

Ejercicio 53
Enunciado: Escribe un programa que reciba una palabra y la imprima en mayúsculas.
Solución:

```
import java.util.Scanner;
public class Ejercicio53 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Palabra: ");
String palabra = sc.nextLine();
System.out.println(palabra.toUpperCase());
sc.close();
}
}
```

Explicación: Se usa el método toUpperCase() para convertir a mayúsculas.

Ejercicio 54
Enunciado: Escribe un programa que imprima los primeros 10 números Fibonacci.
Solución:

```
public class Ejercicio54 {
public static void main(String[] args) {
int a = 0, b = 1;
System.out.println(a);
System.out.println(b);
for (int i = 3; i <= 10; i++) {
int c = a + b;
System.out.println(c);
a = b;
b = c;
}
}
}
```

Explicación: Se calcula la serie sumando los dos números anteriores.

Ejercicio 55
Enunciado: Escribe un programa que reciba un número y muestre su tabla de multiplicar del 1 al 10.
Solución:

```
import java.util.Scanner;
public class Ejercicio55 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
for (int i = 1; i <= 10; i++) {
System.out.println(n + " x " + i + " = " + (n * i));
}
sc.close();
}
}
```

Explicación: Se multiplica el número por valores del 1 al 10.

Ejercicio 56
Enunciado: Escribe un programa que reciba una frase y cuente cuántas palabras tiene.
Solución:

```
import java.util.Scanner;
public class Ejercicio56 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Frase: ");
String frase = sc.nextLine().trim();
if (frase.isEmpty()) {
System.out.println("0 palabras");
} else {
String[] palabras = frase.split("\s+");
System.out.println("Número de palabras: " + palabras.length);
}
sc.close();
}
}
```

Explicación: Se divide la frase por espacios y se cuenta el arreglo.

Ejercicio 57
Enunciado: Escribe un programa que convierta un número decimal a hexadecimal.
Solución:

```
import java.util.Scanner;
public class Ejercicio57 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número decimal: ");
int n = sc.nextInt();
String hex = Integer.toHexString(n);
System.out.println("Hexadecimal: " + hex.toUpperCase());
sc.close();
}
}
```

Explicación: Se usa Integer.toHexString para conversión.

Ejercicio 58
Enunciado: Escribe un programa que imprima los números del 1 al 50 excepto los múltiplos de 7.
Solución:

```
public class Ejercicio58 {
public static void main(String[] args) {
for (int i = 1; i <= 50; i++) {
if (i % 7 != 0) {
System.out.println(i);
}
}
}
}
```

Explicación: Se excluyen números con residuo cero al dividir por 7.

Ejercicio 59
Enunciado: Escribe un programa que reciba un número y muestre la suma de sus cuadrados desde 1 hasta ese número.
Solución:

```
import java.util.Scanner;
public class Ejercicio59 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
int suma = 0;
for (int i = 1; i <= n; i++) {
suma += i * i;
}
System.out.println("Suma de cuadrados: " + suma);
sc.close();
}
}
```

Explicación: Se calcula el cuadrado de cada número y se suma.

Ejercicio 60
Enunciado: Escribe un programa que reciba una cadena y muestre cuántas letras mayúsculas tiene.
Solución:

```
import java.util.Scanner;
public class Ejercicio60 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Cadena: ");
String texto = sc.nextLine();
int contador = 0;
for (int i = 0; i < texto.length(); i++) {
if (Character.isUpperCase(texto.charAt(i))) {
contador++;
}
}
System.out.println("Número de mayúsculas: " + contador);
sc.close();
}
}
```

Explicación: Se cuenta cada carácter que sea mayúscula con isUpperCase.

Ejercicio 61
Enunciado: Escribe un programa que reciba un número y determine si es par o impar.
Solución:

```
import java.util.Scanner;
public class Ejercicio61 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
if (n % 2 == 0) {
System.out.println("Es par");
} else {
System.out.println("Es impar");
}
sc.close();
}
}
```

Explicación: Se utiliza el operador módulo para verificar si el número es divisible por 2.

Ejercicio 62
Enunciado: Escribe un programa que reciba una cadena y la imprima invertida.
Solución:

```
import java.util.Scanner;
public class Ejercicio62 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Cadena: ");
String texto = sc.nextLine();
String invertida = "";
for (int i = texto.length() - 1; i >= 0; i--) {
invertida += texto.charAt(i);
}
System.out.println("Invertida: " + invertida);
sc.close();
}
}
```

Explicación: Se recorre la cadena desde el final hasta el principio concatenando los caracteres.

Ejercicio 63

Enunciado: Escribe un programa que reciba una lista de números separados por espacios y calcule su promedio.

Solución:

```
import java.util.Scanner;
public class Ejercicio63 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Números separados por espacios: ");
String linea = sc.nextLine();
String[] partes = linea.split(" ");
double suma = 0;
for (String parte : partes) {
suma += Double.parseDouble(parte);
}
double promedio = suma / partes.length;
System.out.println("Promedio: " + promedio);
sc.close();
}
}
```

Explicación: Se separan los números, se suman y se divide entre la cantidad para obtener el promedio.

Ejercicio 64

Enunciado: Escribe un programa que reciba un número y muestre su representación en binario.

Solución:

```
import java.util.Scanner;
public class Ejercicio64 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número decimal: ");
int n = sc.nextInt();
String binario = Integer.toBinaryString(n);
System.out.println("Binario: " + binario);
sc.close();
}
}
```

Explicación: Se usa Integer.toBinaryString para convertir a binario.

Ejercicio 65
Enunciado: Escribe un programa que reciba dos números y muestre su máximo común divisor (MCD).
Solución:

```
import java.util.Scanner;
public class Ejercicio65 {
public static int mcd(int a, int b) {
if (b == 0) return a;
return mcd(b, a % b);
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Primer número: ");
int a = sc.nextInt();
System.out.print("Segundo número: ");
int b = sc.nextInt();
System.out.println("MCD: " + mcd(a, b));
sc.close();
}
}
```

Explicación: Se usa el algoritmo de Euclides recursivo para calcular el MCD.

Ejercicio 66
Enunciado: Escribe un programa que reciba un número y muestre su reverso (por ejemplo, 123 → 321).
Solución:

```
import java.util.Scanner;
public class Ejercicio66 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
int reverso = 0;
while (n != 0) {
reverso = reverso * 10 + n % 10;
n /= 10;
}
System.out.println("Reverso: " + reverso);
sc.close();
}
}
```

Explicación: Se extraen los dígitos de derecha a izquierda y se van agregando en orden inverso.

Ejercicio 67
Enunciado: Escribe un programa que imprima la suma de los números impares entre 1 y 100.
Solución:

```
public class Ejercicio67 {
public static void main(String[] args) {
int suma = 0;
for (int i = 1; i <= 100; i += 2) {
suma += i;
}
System.out.println("Suma de impares: " + suma);
}
}
```

Explicación: Se suman solo los números impares usando incremento de 2.

Ejercicio 68
Enunciado: Escribe un programa que reciba una cadena y cuente cuántos caracteres tiene sin espacios.
Solución:

```
import java.util.Scanner;
public class Ejercicio68 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Cadena: ");
String texto = sc.nextLine();
int contador = 0;
for (int i = 0; i < texto.length(); i++) {
if (texto.charAt(i) != ' ') {
contador++;
}
}
System.out.println("Caracteres sin espacios: " + contador);
sc.close();
}
}
```

Explicación: Se cuentan todos los caracteres excepto espacios.

Ejercicio 69
Enunciado: Escribe un programa que reciba una letra y determine si es una vocal o consonante.
Solución:

```
import java.util.Scanner;
public class Ejercicio69 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Letra: ");
char c = sc.next().toLowerCase().charAt(0);
if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
System.out.println("Vocal");
} else if ((c >= 'b' && c <= 'z')) {
System.out.println("Consonante");
} else {
System.out.println("No es letra válida");
}
sc.close();
}
}
```

Explicación: Se verifica si el carácter es una vocal o consonante.

Ejercicio 70
Enunciado: Escribe un programa que calcule el área y perímetro de un rectángulo dados su base y altura.
Solución:

```
import java.util.Scanner;
public class Ejercicio70 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Base: ");
double base = sc.nextDouble();
System.out.print("Altura: ");
double altura = sc.nextDouble();
double area = base * altura;
double perimetro = 2 * (base + altura);
System.out.println("Área: " + area);
System.out.println("Perímetro: " + perimetro);
sc.close();
}
}
```

Explicación: Se usa fórmula área = base * altura y perímetro = 2*(base + altura).

Ejercicio 71
Enunciado: Escribe un programa que reciba un número y muestre si es positivo, negativo o cero.
Solución:

```
import java.util.Scanner;
public class Ejercicio71 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
if (n > 0) {
System.out.println("Positivo");
} else if (n < 0) {
System.out.println("Negativo");
} else {
System.out.println("Cero");
}
sc.close();
}
}
```

Explicación: Se evalúa la condición del número para clasificarlo.

Ejercicio 72
Enunciado: Escribe un programa que reciba un carácter y determine si es una letra mayúscula.
Solución:

```
import java.util.Scanner;
public class Ejercicio72 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Carácter: ");
char c = sc.next().charAt(0);
if (Character.isUpperCase(c)) {
System.out.println("Es mayúscula");
} else {
System.out.println("No es mayúscula");
}
sc.close();
}
}
```

Explicación: Se usa Character.isUpperCase para verificar mayúsculas.

Ejercicio 73
Enunciado: Escribe un programa que imprima los números del 10 al 1 en orden descendente.
Solución:

```
public class Ejercicio73 {
public static void main(String[] args) {
for (int i = 10; i >= 1; i--) {
System.out.println(i);
}
}
}
```

Explicación: Se usa un ciclo for decrementando desde 10 hasta 1.

Ejercicio 74
Enunciado: Escribe un programa que reciba un número y muestre si es un número primo.
Solución:

```
import java.util.Scanner;
public class Ejercicio74 {
public static boolean esPrimo(int n) {
if (n <= 1) return false;
for (int i = 2; i <= Math.sqrt(n); i++) {
if (n % i == 0) return false;
}
return true;
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
if (esPrimo(n)) {
System.out.println("Es primo");
} else {
System.out.println("No es primo");
}
sc.close();
}
}
```

Explicación: Se verifica que el número no tenga divisores entre 2 y la raíz cuadrada.

Ejercicio 75
Enunciado: Escribe un programa que reciba un texto y muestre cuántas vocales contiene.
Solución:

```
import java.util.Scanner;
public class Ejercicio75 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Texto: ");
String texto = sc.nextLine().toLowerCase();
int contador = 0;
for (int i = 0; i < texto.length(); i++) {
char c = texto.charAt(i);
if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
contador++;
}
}
System.out.println("Número de vocales: " + contador);
sc.close();
}
}
```

Explicación: Se recorre el texto contando las vocales.

Ejercicio 76
Enunciado: Escribe un programa que reciba una frase y la imprima sin espacios.
Solución:

```
import java.util.Scanner;
public class Ejercicio76 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Frase: ");
String frase = sc.nextLine();
String sinEspacios = frase.replace(" ", "");
System.out.println("Sin espacios: " + sinEspacios);
sc.close();
}
}
```

Explicación: Se usa replace para eliminar los espacios.

Ejercicio 77
Enunciado: Escribe un programa que reciba una lista de números y muestre el mayor.
Solución:

```
import java.util.Scanner;
public class Ejercicio77 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Números separados por espacios: ");
String linea = sc.nextLine();
String[] partes = linea.split(" ");
int mayor = Integer.MIN_VALUE;
for (String parte : partes) {
int num = Integer.parseInt(parte);
if (num > mayor) {
mayor = num;
}
}
System.out.println("Número mayor: " + mayor);
sc.close();
}
}
```

Explicación: Se compara cada número para encontrar el mayor.

Ejercicio 78
Enunciado: Escribe un programa que reciba un número y muestre su factorial.
Solución:

```
import java.util.Scanner;
public class Ejercicio78 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
long factorial = 1;
for (int i = 2; i <= n; i++) {
factorial *= i;
}
System.out.println("Factorial: " + factorial);
sc.close();
}
}
```

Explicación: Se multiplica sucesivamente desde 1 hasta n para obtener el factorial.

Ejercicio 79
Enunciado: Escribe un programa que reciba una cadena y cuente cuántos dígitos contiene.
Solución:

```
import java.util.Scanner;
public class Ejercicio79 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Cadena: ");
String texto = sc.nextLine();
int contador = 0;
for (int i = 0; i < texto.length(); i++) {
if (Character.isDigit(texto.charAt(i))) {
contador++;
}
}
System.out.println("Número de dígitos: " + contador);
sc.close();
}
}
```

Explicación: Se usa Character.isDigit para contar los dígitos.

Ejercicio 80
Enunciado: Escribe un programa que reciba dos números y muestre su suma, resta, multiplicación y división.
Solución:

```
import java.util.Scanner;
public class Ejercicio80 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número 1: ");
double a = sc.nextDouble();
System.out.print("Número 2: ");
double b = sc.nextDouble();
System.out.println("Suma: " + (a + b));
System.out.println("Resta: " + (a - b));
System.out.println("Multiplicación: " + (a * b));
if (b != 0) {
System.out.println("División: " + (a / b));
} else {
System.out.println("No se puede dividir entre cero");
}
sc.close();
}
}
```

Explicación: Se realizan las operaciones básicas y se verifica división por cero.

Ejercicio 81
Enunciado: Escribe un programa que reciba un número entero y muestre si es múltiplo de 3 y 5.
Solución:

```
import java.util.Scanner;
public class Ejercicio81 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
if (n % 3 == 0 && n % 5 == 0) {
System.out.println("Es múltiplo de 3 y 5");
} else {
System.out.println("No es múltiplo de 3 y 5");
}
sc.close();
}
}
```

Explicación: Se verifica que el número sea divisible tanto por 3 como por 5.

Ejercicio 82
Enunciado: Escribe un programa que reciba una frase y determine cuántas palabras contiene.
Solución:

```
import java.util.Scanner;
public class Ejercicio82 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Frase: ");
String frase = sc.nextLine().trim();
if (frase.isEmpty()) {
System.out.println("Número de palabras: 0");
} else {
String[] palabras = frase.split("\s+");
System.out.println("Número de palabras: " + palabras.length);
}
sc.close();
}
}
```

Explicación: Se usa split con expresión regular para dividir por espacios y contar palabras.

Ejercicio 83
Enunciado: Escribe un programa que reciba un número y muestre su tabla de multiplicar del 1 al 10.
Solución:

```
import java.util.Scanner;
public class Ejercicio83 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
for (int i = 1; i <= 10; i++) {
System.out.println(n + " x " + i + " = " + (n * i));
}
sc.close();
}
}
```

Explicación: Se usa un ciclo para multiplicar el número por 1 hasta 10.

Ejercicio 84
Enunciado: Escribe un programa que reciba un número y muestre si es capicúa (se lee igual al derecho y al revés).
Solución:

```
import java.util.Scanner;
public class Ejercicio84 {
public static boolean esCapicua(int n) {
int original = n;
int reverso = 0;
while (n != 0) {
reverso = reverso * 10 + n % 10;
n /= 10;
}
return original == reverso;
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
if (esCapicua(n)) {
System.out.println("Es capicúa");
} else {
System.out.println("No es capicúa");
}
sc.close();
}
}
```

Explicación: Se compara el número original con su reverso.

Ejercicio 85
Enunciado: Escribe un programa que reciba una lista de números y muestre la suma de los pares.
Solución:

```
import java.util.Scanner;
public class Ejercicio85 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Números separados por espacios: ");
String linea = sc.nextLine();
String[] partes = linea.split(" ");
int suma = 0;
for (String parte : partes) {
int num = Integer.parseInt(parte);
if (num % 2 == 0) {
suma += num;
}
}
System.out.println("Suma de pares: " + suma);
sc.close();
}
}
```

Explicación: Se suman solo los números que son pares.

Ejercicio 86
Enunciado: Escribe un programa que reciba un número y muestre su raíz cuadrada.
Solución:

```
import java.util.Scanner;
public class Ejercicio86 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
double n = sc.nextDouble();
if (n >= 0) {
System.out.println("Raíz cuadrada: " + Math.sqrt(n));
} else {
System.out.println("No se puede calcular la raíz cuadrada de un número negativo");
}
sc.close();
}
}
```

Explicación: Se usa Math.sqrt y se verifica que el número sea positivo.

Ejercicio 87
Enunciado: Escribe un programa que reciba un número entero y muestre la suma de sus dígitos.
Solución:

```
import java.util.Scanner;
public class Ejercicio87 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
int suma = 0;
while (n != 0) {
suma += n % 10;
n /= 10;
}
System.out.println("Suma de dígitos: " + suma);
sc.close();
}
}
```

Explicación: Se extraen y suman cada dígito dividiendo sucesivamente entre 10.

Ejercicio 88
Enunciado: Escribe un programa que reciba un número y muestre la cantidad de dígitos que tiene.
Solución:

```
import java.util.Scanner;
public class Ejercicio88 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
int contador = 0;
if (n == 0) contador = 1;
while (n != 0) {
n /= 10;
contador++;
}
System.out.println("Cantidad de dígitos: " + contador);
sc.close();
}
}
```

Explicación: Se cuenta cuántas veces se puede dividir el número entre 10 hasta llegar a 0.

Ejercicio 89
Enunciado: Escribe un programa que reciba una cadena y muestre si es un palíndromo (se lee igual al derecho y al revés, ignorando espacios).
Solución:
```
import java.util.Scanner;
public class Ejercicio89 {
public static boolean esPalindromo(String texto) {
texto = texto.replace(" ", "").toLowerCase();
int i = 0;
int j = texto.length() - 1;
while (i < j) {
if (texto.charAt(i) != texto.charAt(j)) return false;
i++;
j--;
}
return true;
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Texto: ");
String texto = sc.nextLine();
if (esPalindromo(texto)) {
System.out.println("Es palíndromo");
} else {
System.out.println("No es palíndromo");
}
sc.close();
}
}
```
Explicación: Se compara la cadena de adelante hacia atrás ignorando espacios y mayúsculas.

Ejercicio 90
Enunciado: Escribe un programa que reciba un número y muestre su valor absoluto.
Solución:
```
import java.util.Scanner;
public class Ejercicio90 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
System.out.println("Valor absoluto: " + Math.abs(n));
sc.close();
}
}
```
Explicación: Se usa Math.abs para obtener el valor absoluto del número.

Ejercicio 91
Enunciado: Escribe un programa que reciba un número y muestre si es par o impar.
Solución:

```
import java.util.Scanner;
public class Ejercicio91 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
if (n % 2 == 0) {
System.out.println("Es par");
} else {
System.out.println("Es impar");
}
sc.close();
}
}
```

Explicación: Se usa el operador módulo para determinar si el número es divisible por 2.

Ejercicio 92
Enunciado: Escribe un programa que reciba dos números y muestre cuál es mayor.
Solución:

```
import java.util.Scanner;
public class Ejercicio92 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número 1: ");
int a = sc.nextInt();
System.out.print("Número 2: ");
int b = sc.nextInt();
if (a > b) {
System.out.println(a + " es mayor");
} else if (b > a) {
System.out.println(b + " es mayor");
} else {
System.out.println("Son iguales");
}
sc.close();
}
}
```

Explicación: Se compara ambos números para determinar cuál es mayor o si son iguales.

Ejercicio 93

Enunciado: Escribe un programa que reciba un número y muestre la suma de los números del 1 hasta ese número.

Solución:

```
import java.util.Scanner;
public class Ejercicio93 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
int suma = 0;
for (int i = 1; i <= n; i++) {
suma += i;
}
System.out.println("Suma: " + suma);
sc.close();
}
}
```

Explicación: Se suman todos los números desde 1 hasta n usando un ciclo for.

Ejercicio 94

Enunciado: Escribe un programa que reciba un texto y muestre el texto en mayúsculas.

Solución:

```
import java.util.Scanner;
public class Ejercicio94 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Texto: ");
String texto = sc.nextLine();
System.out.println(texto.toUpperCase());
sc.close();
}
}
```

Explicación: Se usa el método toUpperCase para convertir el texto a mayúsculas.

Ejercicio 95
Enunciado: Escribe un programa que reciba un número y muestre si es múltiplo de 7.
Solución:

```
import java.util.Scanner;
public class Ejercicio95 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
if (n % 7 == 0) {
System.out.println("Es múltiplo de 7");
} else {
System.out.println("No es múltiplo de 7");
}
sc.close();
}
}
```

Explicación: Se verifica que el número sea divisible por 7.

Ejercicio 96
Enunciado: Escribe un programa que reciba un número y muestre la suma de sus números pares desde 0 hasta ese número.
Solución:

```
import java.util.Scanner;
public class Ejercicio96 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
int suma = 0;
for (int i = 0; i <= n; i += 2) {
suma += i;
}
System.out.println("Suma de pares: " + suma);
sc.close();
}
}
```

Explicación: Se suma solo números pares usando incremento de 2.

Ejercicio 97
Enunciado: Escribe un programa que reciba un texto y muestre la cantidad de letras 'a' que contiene.
Solución:

```
import java.util.Scanner;
public class Ejercicio97 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Texto: ");
String texto = sc.nextLine().toLowerCase();
int contador = 0;
for (int i = 0; i < texto.length(); i++) {
if (texto.charAt(i) == 'a') {
contador++;
}
}
System.out.println("Cantidad de letras 'a': " + contador);
sc.close();
}
}
```

Explicación: Se cuenta cuántas veces aparece la letra 'a' en el texto.

Ejercicio 98
Enunciado: Escribe un programa que reciba un número y muestre la suma de los números impares desde 1 hasta ese número.
Solución:

```
import java.util.Scanner;
public class Ejercicio98 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
int suma = 0;
for (int i = 1; i <= n; i += 2) {
suma += i;
}
System.out.println("Suma de impares: " + suma);
sc.close();
}
}
```

Explicación: Se suma solo números impares incrementando de 2 en 2.

Ejercicio 99
Enunciado: Escribe un programa que reciba un número y muestre si es un número perfecto (la suma de sus divisores propios es igual al número).
Solución:
```
import java.util.Scanner;
public class Ejercicio99 {
public static boolean esPerfecto(int n) {
int suma = 0;
for (int i = 1; i < n; i++) {
if (n % i == 0) {
suma += i;
}
}
return suma == n;
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Número: ");
int n = sc.nextInt();
if (esPerfecto(n)) {
System.out.println("Es número perfecto");
} else {
System.out.println("No es número perfecto");
}
sc.close();
}
}
```
Explicación: Se suman los divisores propios y se compara con el número.

Ejercicio 100
Enunciado: Escribe un programa que reciba dos cadenas y determine si son iguales (ignorando mayúsculas y minúsculas).
Solución:
```
import java.util.Scanner;
public class Ejercicio100 {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.print("Cadena 1: ");
String c1 = sc.nextLine();
System.out.print("Cadena 2: ");
String c2 = sc.nextLine();
if (c1.equalsIgnoreCase(c2)) {
System.out.println("Son iguales");
} else {
System.out.println("No son iguales");
```

```
}
sc.close();
}
}
```

Explicación: Se usa equalsIgnoreCase para comparar cadenas ignorando mayúsculas y minúsculas.

Epílogo

Llegar al final de este libro es solo el comienzo de tu viaje en el mundo de la programación con Java. A lo largo de estas páginas, has explorado desde los fundamentos básicos hasta conceptos más avanzados, aprendiendo a pensar como un desarrollador y a resolver problemas con lógica y creatividad.

Java no es solo un lenguaje de programación; es una plataforma que impulsa aplicaciones desde pequeños programas hasta sistemas empresariales complejos, móviles y más allá. La versatilidad y robustez de Java te ofrecen infinitas oportunidades para crecer, innovar y contribuir en un campo en constante evolución.

Recuerda que dominar Java no significa solo conocer su sintaxis o librerías, sino también cultivar el hábito de aprender, experimentar y adaptarte. El código que escribas será un reflejo de tu perseverancia y pasión.

Sigue practicando, construyendo proyectos, leyendo código de otros y manteniéndote actualizado con las nuevas versiones y tendencias. La comunidad de Java es amplia y colaborativa, y siempre habrá espacio para nuevas ideas y soluciones.

Gracias por acompañarme en esta travesía. Ahora, toma tu teclado, abre tu entorno de desarrollo, y crea el futuro que imaginas con Java.

¡Feliz programación!